MASTER IN COMPUTER SCIENCE

Variability Management in Database Applications

Humblet, Mathieu; Tran, Dang

Award date:
2016

Awarding institution:
University of Namur

Université de Namur
Faculty of Computer Science
Academic Year 2015–2016

Variability Management in Database Applications

Mathieu Humblet Dang Vinh Tran

Internship mentor: Jens Weber

Supervisor: _____ (Signed for Release Approval - Study Rules art. 40)
Anthony Cleve

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur

Abstract

Complex software products are often subject to context specific configurations and variations. Variability management is a critical aspect of engineering software product lines efficiently. However, the variability management on the data aspect of the systems has not been largely explored.

In this thesis, we have a contribution in three parts. First of all, we introduce a specific methodology to implement variability in database applications which enables software engineers to model feature models, map them to a database schema elements, and finally produce a new database schema including only the selected features. Then we present the Simple Variability Language, a language we designed for this specific purposes. Finally we present SVL Tool, a plug-in for the CASE Tool DB-MAIN that implements our methodology and language. We also present our results on OSCAR, an Electronic Medical Records software program widely used in Canadian clinical settings.

We summarized our work in a paper entitled "Variability Management in Database Applications", that was accepted as a submission for the First International Workshop on Variability and Complexity in Software Design (VACE), a workshop of the 38th International Conference on Software Engineering (ICSE 2016).

Résumé

Les produits logiciels complexes sont souvent soumis à des configurations et des variations spécifiques au contexte. La gestion de la variabilité est un aspect essentiel à l'ingénierie efficace de lignes de produits logiciels. Cependant, la gestion de la variabilité en ce qui concerne les données des systèmes est un champ de recherche qui est loin d'avoir été largement exploré.

Dans ce mémoire, nous apportons une contributon en trois parties. Tout d'abord, nous présentons une méthodologie spécifique pour implémenter la variabilité dans des applications de bases de données, méthodologie qui qui permet aux ingénieurs logiciels de modéliser des modèles de fonctionnalités, de les lier avec les éléments de schémas de base de données, et finalement de produire un nouveau schéma de base de données. Nous présentons ensuite le Simple Variability Language, un langage que nous avons conçu dans ce but précis. Enfin, nous présentons SVL Tool, un plug-in pour le CASE Tool DB-Main qui implémente notre méthodologie et langage. Nous présentons également nos résultats sur OSCAR, un système de dossier médicaux électroniques largement utilisé dans les établissements cliniques du Canada.

Nous avons résumé notre travail dans un article intitulé "Variability Management in Database Applications", qui a été accepté comme soumission pour le premier International Workshop on Variability and Complexity in Software Design (VACE), un des workshops de la 38ème édition de l'International Conference on Software Engineering (ICSE 2016).

Acknowledgments

Throughout this work, we received help from many people and we would like to thank all of them for the precious time and support they offered.

First and foremost, would like to express our sincere gratitude to Anthony Cleve, our thesis supervisor, for his support throughout this project, for the time he dedicated to us before, during and after our internship and finally for his motivation, enthusiasm and his expertise. We also would like to acknowledge him for his trust, as he gave us the opportunity to present our work at the *International Conference on Software Engineering*, one of the largest annual software engineering conferences in the world.

Then we would like to thank our internship mentor Jens Weber, who has shown a large and consistent interest in our project. Without him and Anthony we probably could not have been able to realize this work. Our numerous scientific discussions and their constructive comments have greatly improved this work.

Part of the thesis work was done while we were visiting students at the University of Victoria, British Columbia. It is a pleasure to thank the Simbioses Lab for their hospitality. We would also like to acknowledge Morgan Price for his encouragement and his support during our stay. It is thus a great pleasure to formally thank Paule Bellwood, Siemon Diemert, Jeremy Ho, Anita Katahoire, Fieran Mason-Blakley and Raymond Rusk, our Simbioses lab mates for their help and the good company during the months we spent in their office. A very particular thanks goes to Ryan Habibi for his warm and hospitable welcome from the first day to the last.

Besides the lab, we are also indebted to the many people in Victoria in and outside the lab who made our stay in Canada a very pleasant one. We would thus like to thank our landlady for having welcomed us into her house during this period, and our many housemates that instantly made us feel at home during these four months abroad.

Last but not least, we wish to address our gratitude to our friends and family who supported us during our time there. We would never have been able to finish this thesis without their support.

Contents

Glossary

DBMS is a program that provides functions to define, exploit and management of databases and their content [33].

Drupal An open source content management platform that supports a variety of Web sites ranging from personal Weblogs to large community-driven Web sites [20] .

ETL ETL (Extract, Transform and Load) is a process in data warehousing responsible for pulling data out of the source systems and placing it into a data warehouse [26].

Java Servlet A small program that runs on a server. The term usually refers to a Java applet that runs within a Web server environment. [63] .

JavaScript A scripting language developed by Netscape to enable Web authors to design interactive sites [39].

Linux The Linux open source operating system, or Linux OS, is a freely distributable, cross-platform operating system based on Unix that can be installed on PCs, laptops, netbooks, mobile and tablet devices, video game consoles, servers, supercomputers and more [72] .

MySQL MySQL is an open source relational database management system [48] .

OSCAR The OSCAR System stores Electronic Medical Records (EMRs) designed to help improve health care from individual to population health levels while reducing costs.

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction & Motivation

In this chapter we will introduce the subject and the motivation of our work, as well as detail the structure of this thesis.

1.1 Variability as a Complex Issue of Today

Nowadays, software systems are getting more and more complex. Actually, they have always been, as it is a trend that has been present as soon as the first software systems were being written, and soon overtook by their younger counterparts. However, today the complexification of software is obvious even for the young experts.

There may be many explanations for this evolution, as for instance the never-ending race for the fastest hardware, allowing computers to carry out more difficult tasks, the changes in the infrastructure architectures, evolutions in the user needs or even in the users workflow itself. The intent of this work is not the understand why software systems are more complex today than one or ten years ago, but precisely to study how this complexity can be broken down to fulfill in the best possible way the user needs.

We now need to give a first and broad definition of software product lines, which are a way to solve this complexity issue. Software product lines can thus be broadly defined as a family of multiple versions of a single basic system, each version being tailored for specific needs, and different from the other versions on specific points.

Starting from there, we can give a first glimpse of the notion of software variability. We will define it more precisely later, but we can start with the assumption that it is the change prone characteristic of software product lines. We can instinctively see that variability is the main concept of software product line engineering as it represents the variations between two versions of the same software product line. Furthermore, variability is a complex matter, as it has to be modeled, managed, tested, extracted, and documented.

1.2 Variability in Databases

Since the beginning of software engineering, databases have been seen as a core component of the software. They can be seen as containers for the software data, and they do that in a way that can be optimized and analyzed. Data itself can be seen as the primary resource for most of the computing tasks, explaining the importance of databases. And as every other software component, databases are also subject to variability.

Indeed, databases today have become enormous systems on their own, often hosted on their own hardware machines that are specialized in giving access to the within the shortest time possible. Moreover, the overall performance of a software system usually relies on the database access time, as hard drives are the software components that are seen as the bottlenecks of the system.

In this regard, the study of variability in databases makes perfect sense. If a user can have a tailored version of his database, he can save space, time and computational power compared to using a generic database.

1.3 Research Questions

With this context in mind, we can now formulate our main research question which is "How to represent and relate variability models in database applications?"

From this main question can be deduced relative sub-questions, such as "how to help a software reengineering expert to implement variability in database applications?" or "is it possible to develop a whole methodology to implement variability at the database level?" These questions have guided us all along our work, as we detail in the following chapters.

1.4 Thesis Structure

Chapter 2 contains the technological background required to fully understand this thesis. This knowledge is provided for everyone, but especially for readers that are not familiar with the technical concepts covered in our work.

Chapter 3 presents the state of the art on our research subject. We detail here our literature review methodology and detail our results.

Chapter 4 introduces briefly the context of our work on OSCAR EMR, an Electronic Medical Records system used widely across Canadian health facilities.

Chapter 5 explains our methodology to study the subject, and breaks it down to the different processes we conducted.

Chapter 6 contains our design processes. We detail here the models of our tool, such as the algorithms, data structures or meta models we created.

Chapter 7 is a presentation of the actual tool we developed. Using its interface, we describe all its features and how the user is supposed to use them.

Chapter 8 is about the case study we made using our tool on OSCAR EMR. We first explain the OSCAR system in detail, and then provide the results we obtained by applying our tool to OSCAR.

Chapter 9 is an additional discussion about our work. It is the opportunity for us to bring some hindsight to our work, by giving an overview of the qualities and defects of our work and how they can be dealt with.

Chapter 10 concludes the thesis.

Chapter 11 opens perspective about the future works and applications.

Chapter 2

Technological background

2.1 Relational Databases

In this section we present important concepts and terms needed to understand relational databases

2.1.1 Fundamental Concepts

We will use the definitions as they have been written and explained by Codd in [16].

Domain A domain is a set of values, each of them referring to an object of the real world.

Relation Starting from the mathematical definition of relation, Codd says that "When conceived as a table, R has the following properties" : first, each row represents a tuple of R; then, the ordering of rows is immaterial; finally, all rows are distinct from one another in content.

Relation is also the only compound data type, it means that it can be decomposed by the DBMS. It also means that "Whatever is conceived as entities, and whatever is conceived as relationships, are perceived and operated upon in the relational model in just one common way: as relations".

Finally, we can say that a relation is a named set of aggregates of n values, each value belonging to a domain. A **row** is such an aggregate, when an **attribute** is formed by the components of same rank of the rows. For the **table** term, it is defined later as synonymous of "relation" (actually, they should be called "R-tables", but an abuse of language leads to just "tables". On the same note, an attribute corresponds to a **column**, and a tuple to a **row**).

Identifier and primary key On a mathematical point of view, an identifier of a relation is a subset of its attributes so that there cannot exist at any time more than one line having the same values of these attributes. If we look at the relational equivalence, we see that each table has exactly one primary key. This key is a combination of columns such that, on the first hand, the value of the primary key in each row of the pertinent R-table identifies that row uniquely and that, on the other hand, if the primary key is composite and if one of the columns is dropped from the primary key, the first property is no longer guaranteed.

Foreign key We have seen that the value of a primary key in each row of a table defines the particular object represented by that row uniquely within the type of objects that are represented by that relation. That means that if everywhere else in the database that there is a need to refer to that particular object, the same identifying value drawn from the same domain is used. Any column containing those values is called a **foreign key**.
Another way to see this concept is through the definition of referential integrity, a property of foreign keys: Let D be a domain from which one or more primary keys draw their values. Let K be a foreign key, which draws its values from domain D. Every unmarked value which occurs in K must also exist in the database as the value of the primary key on domain D of some base relation.

2.1.2 Overview of the Database Design Process

We can now explain the whole process of designing a database. These explanations are translated from the work of Hainaut [33].

The figure 2.1 details the successive steps to design a database. An explanation for each of them follows in the next paragraph.

The first part of the database design process is the **Conceptual analysis**, which is the analysis of the user requirements to produce conceptual schemas, expressed in the entity-relationship model. It is followed by the **Logical design**, which is a translation of the conceptual schema into the model of the DBMS. After that, the **Physical design** is an enhancement of the logical schema, especially performance-wise, thanks to technical constructions. The enhanced schema is the physical schema. Finally, the **Code production** is the translation of the physical schema into DLL code (which is explained in the section 2.1.5).

2.1.3 Conceptual schemas and the Entity-Relationship Model

The notions defined in this section and the following ("Physical schemas") are taken from the book of Hainaut [33].

A conceptual schema is a formal representation of the system user requirements that describes the relevant static concepts of the application domain. To create a conceptual schema, we use the entity-relationship model, which contains the following constructions : first of all, an **entity type** is a class of domain entities; then an **attribute** is a characteristic of an entity type; finally, an **association type** is a type that defines all the associations between the same entity types.

2.1.4 Logical schemas

A logical schema is the translation of a conceptual schema according to the model of a DBMS. It is created using the logical relational model which is defined as following:

- the schema contains entity types (named *tables*)
- every entity type has at least one attribute (named *column*)
- an attribute is mono-valued and atomic, it is mandatory or optional
- the only constraints are the ones induced by identifiers and foreign keys

2.1.5 The SQL Language

The SQL (structured query language) language is a data sublanguage which was invented in IBM Research in 1972. ([16]). The SQL can be distinct in two sublanguages :

1. SQL DDL (for Data Definition Language) is used to define, delete or modify a table, a domain, a column of a constraint
2. SQL DML (for Data Modification Language) is used to extract data from a table.

2.2 CASE Tools

CASE[1] tools are a set of application programs in order to help development and maintenance of software projects [68]. They are used to ease the organization and control the development of software, especially on large, complex projects involving many software components and people [57]. Baik et al. states that since their emergence in the 1970s, CASE tools ave played a critical role in improvement of software productivity and quality by assisting tasks in software development processes [9]. As McMurtrey et al. explain in the introduction of their study on the real world use of CASE tools, "CASE tools possess many features and functionalities that contribute to improving the performance of systems analysts and designers" [43] (the advantages and drawbacks of CASE tools will be detailed in a later paragraph).

[1] acronym for Computer Aided Software Engineering

The importance of the CASE tools can be measured through the market expansion during the 1990s: Jarzabek and Huang the annual worldwide market for CASE tools was $4.8 while it grew to approximately $12.11 billion in 1995 [38]. With such a growing demand, from the business world, several CASE tools were developed. As far as the adoption of such tools is concerned, Kemerer studied the question of their adoption. They conclude their work with a list of factors such as the education level of the staff, the culture of the organization and the training that goes with the implementation of the CASE Tool [40].

2.2.1 Type of CASE Tools

There are a lot of different CASE Tools. This subsection presents some of these categories with a short explanation (all of these come from [68]).

- Diagram tools: Diagram tools are used to graphically represent components, data and control flow of the system among various software components and system structure, generally without effort. These tools allow creating a flowchart which is a type of diagram that represents an algorithm, workflow or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows.
- Process Modeling Tools: Process modeling is a method to create software process model, which is used to develop the software. Process modeling tools help the managers to choose a process model or modify it like the requirement of software products. For example, EPF Composer is a framework for Eclipse. This framework aims at creating a customizable software process and supports a large variety of types of project [21].
- Project Management Tools: Project Management Tools are used for project planning, cost and effort estimation, project scheduling and resource planning. In project management, the execution of every mentioned step in software project management must be strictly respected by managers. Moreover, these tools help in storing and sharing project information in real-time throughout the organization. For example, GanttProject is a free project management software [31].
- Documentation Tools: Documentation in a software project contains all information of all phases of the system development life cycle from the beginning to the end of the project. Documentation tools are used to generate documents for (technical- and end-) users. Technical users documents refer to system manual, reference manuals, training manuals, installation manuals, etc. because these users are generally in the development team. The end user documents describe the functioning and how-to of the system such as user manual for everyone who will use it. For example, Doxygen is a free licensed documentation generator able to produce software documentation from the source code of a program [19].
- Analysis Tools: Analysis tools are used to collect requirements from all placeholders of a project and to automatically check if there are any inconsistencies, data redundancies, inaccuracies in the diagram, etc. There are different tools for different purposes, for example, Accompa is used for requirement analysis whereas

Visible Analyst is used for a total analysis.
- Design Tools: Design tools are used by the designer to conceptualize the structure of the software. It exists a "Computer-aided design" (CAD) to help a designer to create, modify, analyze or optimize the design. There is a plethora of design tools.
- Programming Tools: Programming tools are computer programs used by developers to create, modify, maintain, test, etc the source code of a project. These tools consist of programming environments like an integrated development environment (IDE). For example, Eclipse which is an essential tool for any Java developer [Ecl, 2015].
- Prototyping Tools: A prototype is used to provide an initial look of the product and how it will work by simulating a few functionalities. Prototypes have a different level of fidelity (low-, medium-, high-). Typically, the low-fidelity prototype is a paper prototype, focused on the interaction aspect. The medium-fidelity prototype is a clickable wireframe and the high-fidelity prototype is a faithful representation of the interaction behavior and the look and feel of the system. This prototype can be fully programed. Prototypes tools are used to help a team to quickly build prototypes due to the existing information. For example, Mockup Builder is a wireframe and create a prototype from requirements [46].
- Web Development Tools: Web Developments tools are used to help web developers to test and debug their source codes. Moreover, these tools assist web developers to design their interface. They can directly see what they are developing. For example, Fontello, Foundation 3 are Web Developments tools.
- Quality Assurance Tools: Quality assurance in a software organization is any process of checking to see if the software product being developed complies with requirements and organization standards [56]. Quality Assurance Tools is a set of control and software testing tools. For example, AppsWatch is a testing tool for performance, a web testing, etc. [5].
- Maintenance Tools: After the delivery, software needs to be maintained and can need some modifications. Logging, error reporting techniques, etc. are useful for the maintenance of software. Maintenance tools provide these techniques.
- Data Modeling Tools: Data modeling is often the first step in database design due to an analysis of data objects and their relationships to other data objects [71]. So, designers create a conceptual schema of data objects related to each other. The data modeling of a database starts from a conceptual model to a logical model to a physical schema. Data modeling tools are used to help designers to do this. For example, DB-MAIN is a free tool for data modeling and data architecture (See the next section to have more information about it).

This list of tools is not complete but provides an indication of the type of existing tools.

2.2.2 Advantages and Disadvantages

The qualities and defects of CASE Tools listed here come from [3].

Advantages

- Increased Speed: CASE Tools provide automation and reduce the time to complete many tasks, especially those involving diagramming and associated specifications. Estimates of improvements in productivity after application range from 355 to more than 200%.
- Increased Accuracy: CASE Tools can provide ongoing debugging and error checking which is very vital for early defect removal, which actually played a major role in shaping modern software.
- Reduced Lifetime Maintenance: As a result of better design, better analysis and automatic code generation, automatic testing and debugging overall systems quality improves. There is better documentation also. Thus, the net effort and cost involved with maintenance is reduced.
- Better Documentation: By using CASE Tools, vast amounts of documentation are produced along the way. Most tools have revisions for comments and notes on systems development and maintenance.
- Programming in the hands of non programmers: With the increased movement towards object oriented technology and client server bases, programming can also be done by people who don't have a complete programming background.
- Intangible Benefits: CASE Tools can be used to allow for greater user participation, which can lead to better acceptance f the new system. This can reduce the initial learning curve.

Disadvantages

- Tool Mix: It is important to make an appropriate selection o tool mix to get cost advantage. CASE integration and data integration across all platforms is also very important.
- Cost: CASE is not cheap, as most firms engaged in software development on a small scale do not invest in CASE tools because they think that the benefits of CASE are justifiable only in the in the development of large systems. The cost of outfitting every systems developer with a preferred CASE tool kit can be quite high. Hardware and systems, software, training and consulting are all factors in the total cost equation.
- Learning Curve: In most cases, programmer productivity may fall in the initial phase of implementation, because users need time to learn the technology. In fact, a CASE consulting industry has evolved to support uses of CASE tools. The consultants offer training and on-site services that can be crucial to accelerating the learning curve and to the development and use of the tools.

2.3 DB-MAIN

Before the development of DB-MAIN [25], CASE tools often provided partial solutions to the engineering problems of the databases. There were some weaknesses in those

tools, e.g., ignorance of non-functional requirements, a lack of flexibility, ignorance of some processes, etc. The idea of a tool such as DB-MAIN was born from these gaps, to have a tool which will pay attention to non-functional requirements such as optimization, to flexibility and some key processes such as maintenance.

DB-MAIN is a free tool for data modeling and data architecture [**DB-MAIN**].This tool is developed in C++ with the widget toolkit "wxWidgets" and runs on Windows, Linux and Mac OSX. The purpose of this tool is helping developers and analysts in different data engineering processes. Here is a list of these processes:

- Design processes: requirement analysis, conceptual design, normalization, schema integration, logical design, physical design, schema optimization, code generation.
- Transformations: schema transformation, model transformation, ETL.
- Reverse engineering and program understanding: schema analysis (COBOL, CODASYL, IMS, IDMS, SQL, XML, ...), code analysis, data and data flow reverse engineering.
- Maintenance, evolution and integration: database migration, database evolution, impact analysis, database integration and federation, data wrapper design and generation.
- And many other domains like temporal and active databases, data warehouse, XML engineering, ...

DB-MAIN allows users to develop new functions and extend its repository due to possibilities to add new plugins. Moreover, SVL Tool contains a plethora of functions used in data modeling, such as code generator (SQL, MySQL, etc.) and a JDBC extractor (which extracts database structures to a JDBC driver), etc.

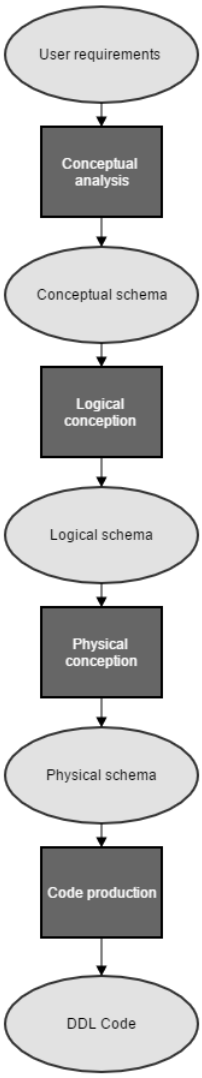

Figure 2.1: Database Design Process, adapted from [33]

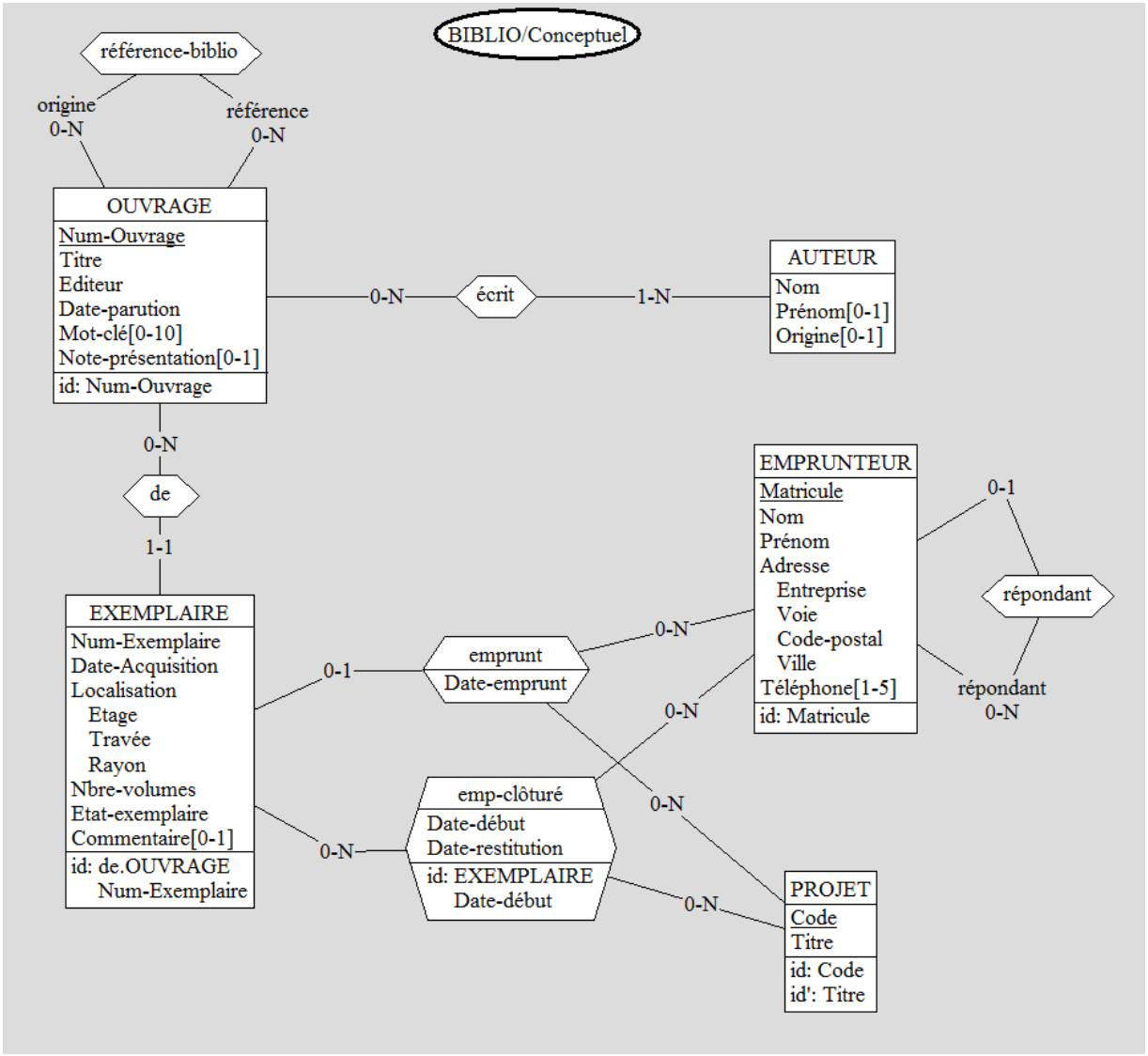

Figure 2.2: Example of conceptual schema, DB-MAIN documentation

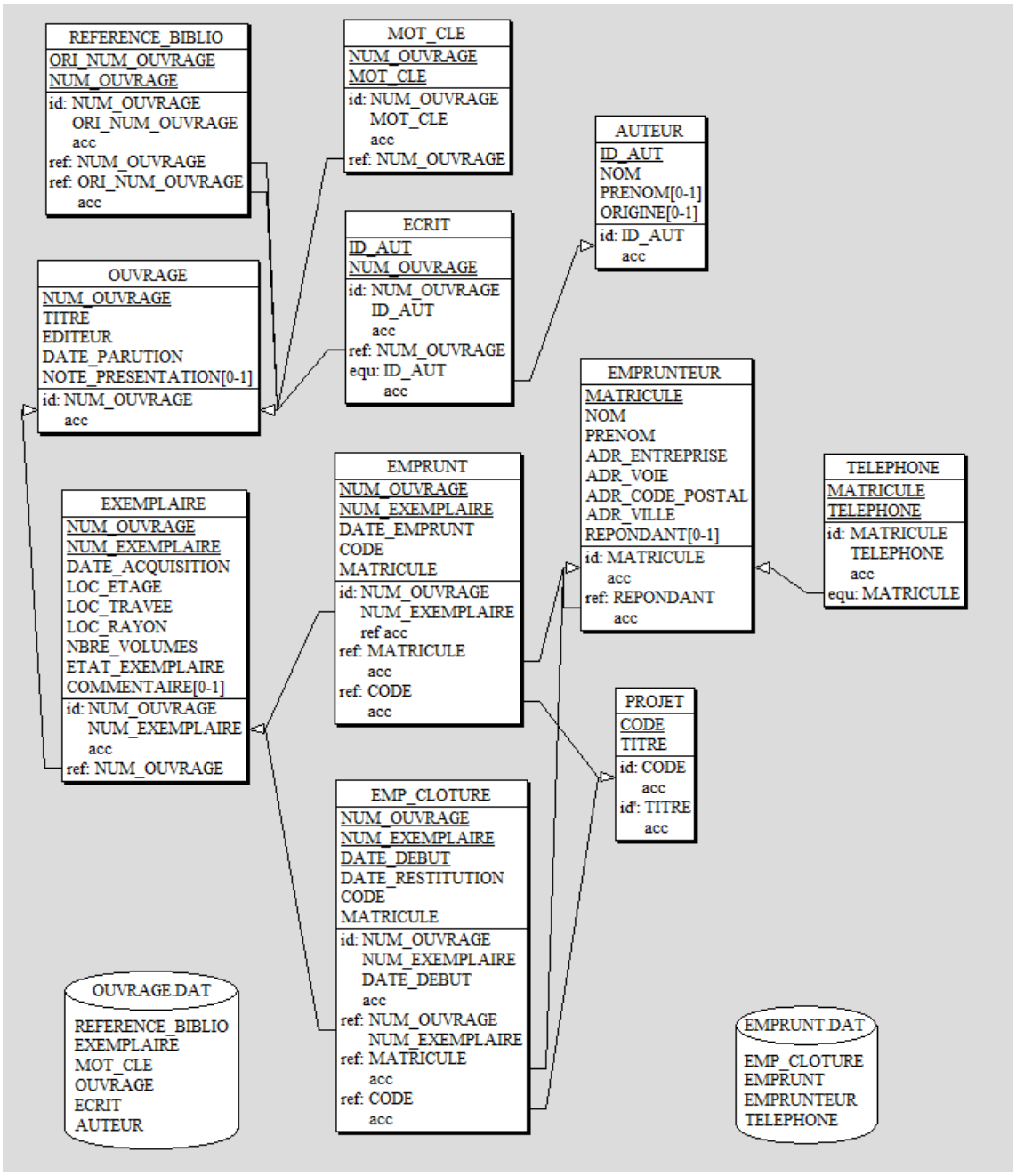

Figure 2.3: Example of physical schema, DB-MAIN documentation

Chapter 3

State of the art

In this section we provide an overview of the current state of the art on software product lines and the different aspects of variability.

3.1 Literature selection

We first have to detail our methodology to survey the literature.

3.1.1 Inclusion criteria

Have been included in our review the papers who meet the following criteria : the documents had been published between the 1st of January 2001 and the 1st of September 2015 for relevance reasons, even if we also allowed older references that are considered fundamental (important number of citations), they were falling in the domains of variability and/or database management (one of the two being enough for a document to be accepted), written in English and they could be books, systematic reviews, surveys, case studies.

3.1.2 Research strategy

To carry out our study, we primarily focused on the two most important information sources in Computer Science: the Institute of Electrical and Electronics Engineers (IEEE) digital library and the Association for Computing Machinery (ACM) digital library.

Our base requests are the following keywords : "variability management" OR " database applications " OR " software product lines "

3.2 Software product lines

3.2.1 Motivation

As we can read in [69], "the improvement of costs and time to market are strongly correlated in software product line engineering: the approach supports large-scale reuse during software development. As opposed to traditional reuse approaches, this can be as much as 90% of the overall software. Reuse is more cost-effective than development by orders of magnitude. Thus, both development costs and time to market can be dramatically reduced by product line engineering".

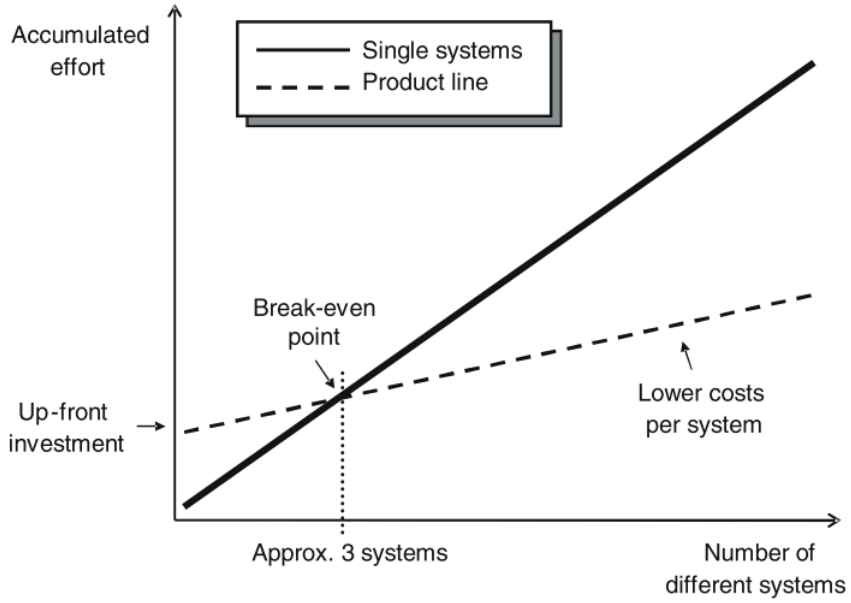

Figure 3.1: Economics of software product line engineering, [69]

3.2.2 Software product lines

Definition A software product line is defined by the Software Engineering Institute (Carnegie Mellon University) as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."

Software product line engineering

– Platform : a platform is any base of technologies on which other technologies or processes are built.

– Mass customization : is the large-scale production of goods tailored to individual customers' needs.

We can now introduce a definition of software product lines :

"Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation". [54]

Software Product Line Engineering lifecycle

"Application engineering is strongly driven by the product line infrastructure, which usually contains most of the functionality required for a new product. The variability explicitly modeled in it provides the basis for deriving the individual products. Basically, when a new product is developed, an accompanying project is set up. Then requirements are gathered and directly categorized as being part of the product line (i.e. a commonality or variability) or product-specific. Then the various assets (e.g. architecture, implementation, etc.) may be instantiated right away, leading to an initial product version. At this stage in the development, up to 90% of the product may be available from reuse; only the remaining 10% must be developed in further steps." [69]

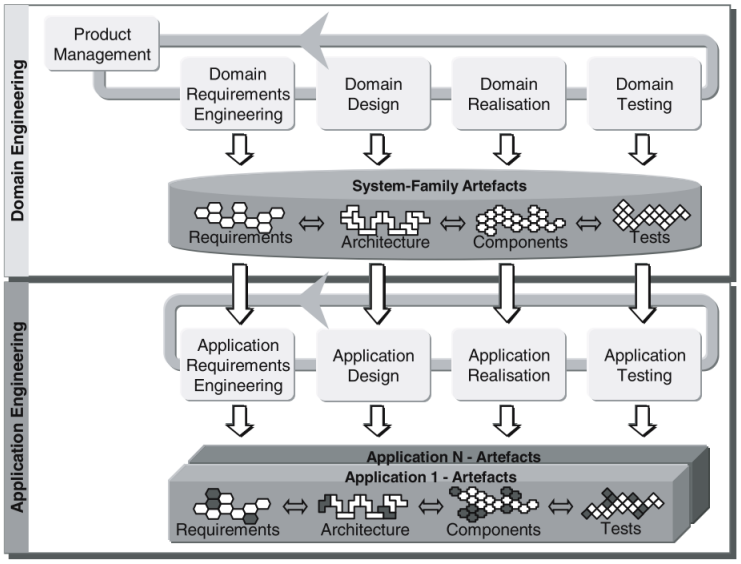

Figure 3.2: The two-lifecycle model of software product line engineering, [69]

3.3 Variability

3.3.1 Variability concepts

Variability is an essential concept for software product lines development. We need to define the following concepts to really understand what is variability (the following definitions, as the other ones from this subsection, are from [54])

- Variability Subject : A variability subject is a variable item of the real world or a variable property of such an item.
- Variability Object : A variability object is a particular instance of a variability subject. Consciousness
- Variation Point : A variation point is a representation of a variability subject within domain artifacts enriched by contextual information.
- Variant : A variant is a representation of a variability object within domain artifacts.

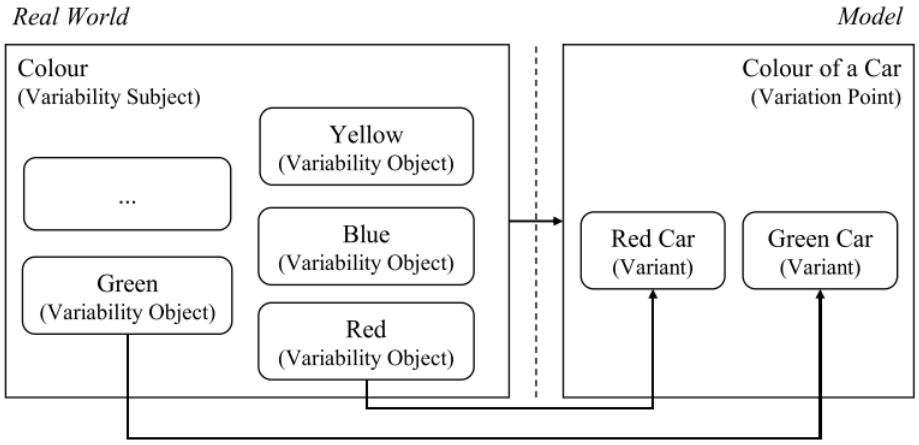

Figure 3.3: Relation between variability in the real world and in a model of the real world, [69]

There are other aspects important to the variability, such as:

- Variability in time is the existence of different versions of an artifact that are valid at different times.
- Variability in space is the existence of an artifact in different shapes at the same time.
- External variability is the variability of domain artifacts that is visible to customers.

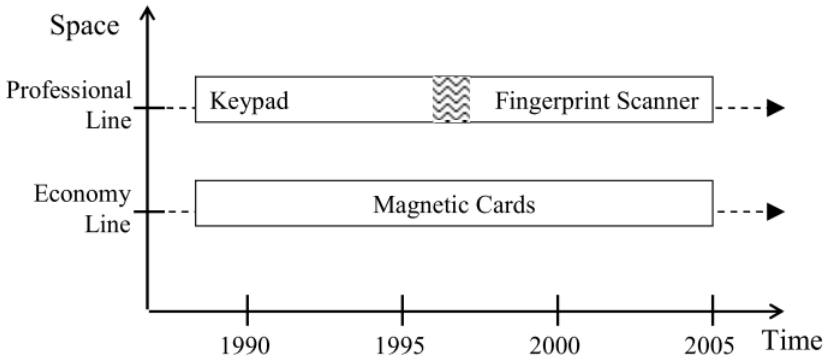

Figure 3.4: Variability in time and space, [69]

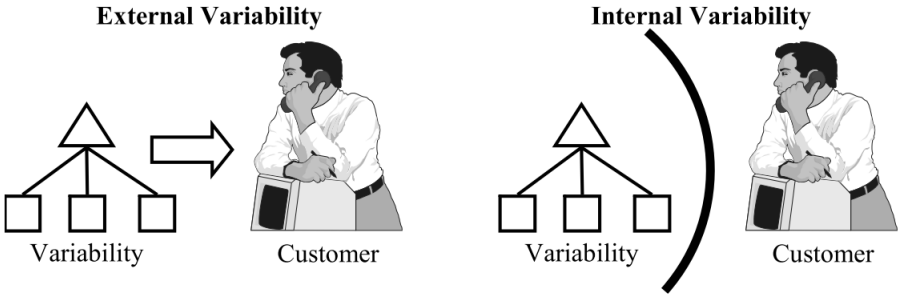

Figure 3.5: Internal and External Variability, [69]

– Internal variability is the variability of domain artifacts that is hidden from customers.

Finally, we need to define the dependencies that may exist in variability :

– Variant Constraint Dependency :
  – Variant requires variant : The selection of a variant V1 requires the selection of another variant V2 independent of the variation points the variants are associated with
  – Variant excludes variant : The selection of a variant V1 excludes the selection of the related variant V2 independent of the variation points the variants are associated with.
– Variant to Variation Point Dependency : The variant to variation point constraint dependency describes a relationship between a variant and a variation point, which may be of one of the two types:
  – Variant requires variation point (requires V VP): The selection of a variant V1 requires the consideration of a variation point VP2.

  - Variant excludes variation point (excludes V VP): The selection of a variant V1 excludes the consideration of a variation point VP2
- Variation Point Constraint Dependency
  - Variation point requires variation point (requires VP VP): A variation point requires the consideration of another variation point in order to be realized.
  - Variation point excludes variation point (excludes VP VP): The consideration of a variation point excludes the consideration of another variation point.

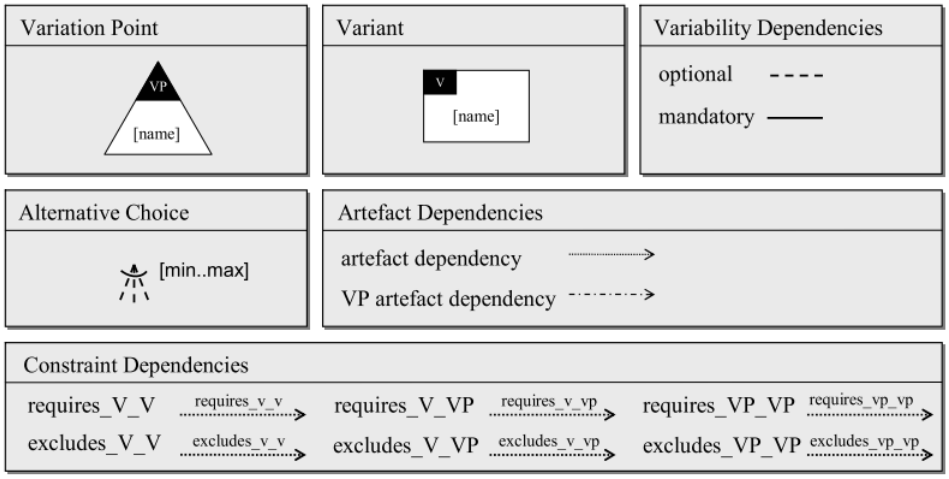

Figure 3.6: Graphical notation for variability models, [69]

To conclude this first section about variability, we can quote Pohl et al. about the link between variability and software product lines: "Variability of a software product line is variability that is modeled to enable the development of customized applications by reusing predefined, adjustable artifacts." [54]

3.4 Domain engineering

3.4.1 Feature Oriented Domain Analysis

As we know that our work will deal with the modeling of features, Feature Oriented Domain Analysis (FODA), a popular approach for this purpose, seems relevant to our work.

Feature

First of all, we have to define what a feature is. According to Apel et al. in [4], "A feature is a characteristic or end-user-visible behavior of a software system. Features

are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle".

Feature and Software Product Line Engineering

We can see the importance of FODA in software product lines engineering as, still according to Apel et al. in [4], "features are the concerns of primary interest in product-line engineering."

Berger et al. establish in [12] a qualitative study of features in industrial software product lines, based on an analysis of 23 features uses in industrial settings by three large companies.

Chen and Babar make a systematic review in [15] and make the statement that "proposed approaches may be very beneficial when they are applied properly in appropriate situations".

Apel et al. also explains a process in software product line engineering (see figure 3.7). They also define four clusters of tasks in product-line development:

- Domain analysis is a form of requirements engineering for an entire product line. Here, we need to decide the scope of the domain, that is, decide which products should be covered by the product line and, consequently, which features are relevant and should be implemented as reusable artifacts. The results of domain analysis are usually documented in a feature model.
- Requirements analysis investigates the needs of a specific customer as part of application engineering. In the simplest case, a customer's requirements are mapped to a feature selection, based on the features identified during domain analysis. If novel requirements are discovered, they can be fed back into domain analysis, which may result in a modification of the feature model (and the reusable domain artifacts).
- Domain implementation is the process of developing reusable artifacts that correspond to the features identified in domain analysis. Although there are many kinds of artifacts relevant in software product lines (including design, test, and documentation artifacts)
- Product derivation (or product generation or product configuration or product assembly) is the production step of application engineering, where reusable artifacts are combined according to the results of requirement analysis. Depending on the implementation approach, this process can be more or less automated, possibly, involving several development and customization tasks.

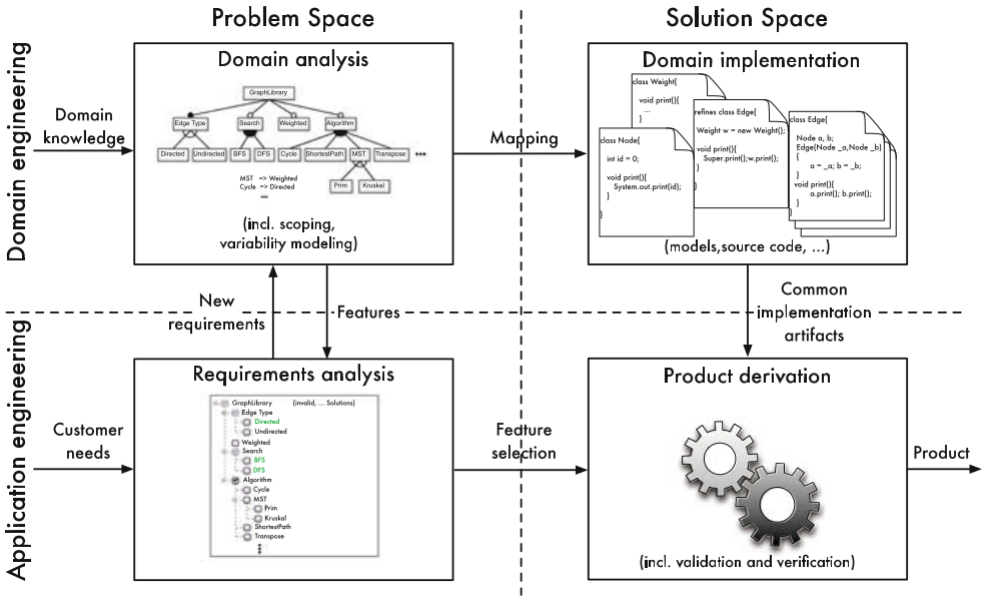

Figure 3.7: Overview of an engineering process for software product lines, [4]

Feature extraction

Dit et al. conducted in [18] a systematic review of feature location techniques. Eighty-nine articles from 25 venues have been reviewed and classified within the taxonomy in order to organize and structure existing work in the field of feature location.

Lee et al. provide in [42] report on using a Domain Model-Based Extractive Approach to Software Product Line Asset Development. They developed a domain model-based extractive method which has been applied to introduce software product lines to a set-top box manufacturing company.

3.4.2 Delta Oriented Domain Analysis

As seen by Schaefer et al. in [62], "a product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing code. Application conditions attached to delta modules allow handling combinations of features explicitly. A product implementation for a particular feature configuration is generated by applying incrementally all delta modules with valid application condition to the core module. In order to evaluate the potential of DOP, we compare it to FOP, both conceptually and empirically."

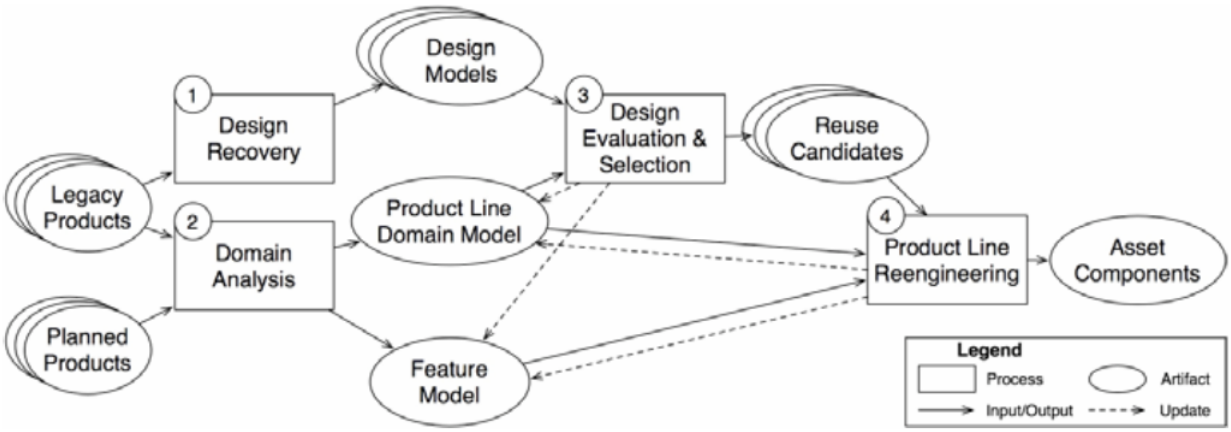

Figure 3.8: Engineering process, [42]

3.5 Variability Modeling Languages

Now that we have defined variability, we can explain the different ways to create variability models.

3.5.1 Variability modeling languages overview

Berger and al. review the variability modeling techniques used in the industry in [11].

For the textual variability languages, the review of [22] analyses systematically the textual variability languages, with amongst them TVL, CVL and IVML.

INDENICA Variability Language IVML is a modeling language defined in [23]. Examples of use are provided in [24].

Common Variability Language and Base Variability Resolution The Common Variability Language (CVL) is another one, which specification can be found in [37]. Several case studies of CVL can be found. In [8] it is applied to a process variability management. In [53] we see the Departement of Defense using it for the definition of a framework. Rouille also uses it to manage variability in software process lines in [55]. Another interesting case is the software product line development for train control detailed in [65]. Last but not least, in [27] CVL is used to transform Java code.

The Base Variability Resolution (BVR) is the new version of the CVL. Figure 3.10 shows a top level overview of BVR.

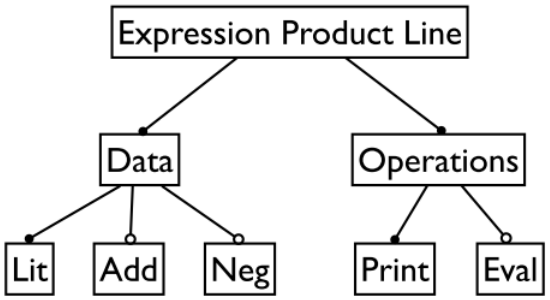

Figure 3.9: FeatureModel for Expression Problem Product Line

Textual Variability Language TVL is a textual variability language designed by the University of Namur.

3.6 Variability management

For the management of variability, Chen and Ali Babar conducted a systematic review of the evaluation of variability management approaches in [14]. They review 97 papers and their general conclusion is that "that the status of evaluation of VM approaches is unsatisfactory rather poor in certain areas".

Metzger and Pohl gives an overview of the achievements and challenges of variability management in [45]. They analyze more than 600 documents in order to produce a synthesis over different aspects of variability.

- Foundations : two product line process, product line variability, product line variability vs software variability
- Variability modeling and analysis : modeling product line variability, analyzing variability
- Domain engineering : product management, domain requirements engineering, domain design, domain realization, domain quality assurance
- Application engineering : requirements, design, realization and quality assurance
- Emerging challenges : variability management in non product line settings, leveraging instantaneous feedback, open world assumption

Pohl et al. distinct two principle ways to model variability : first, the "Integrated Documentation": in order to support the integrated documentation of product line variability, dedicated or specialized modeling and documentation concepts are introduced into existing modeling languages or document templates. Then, the second principle

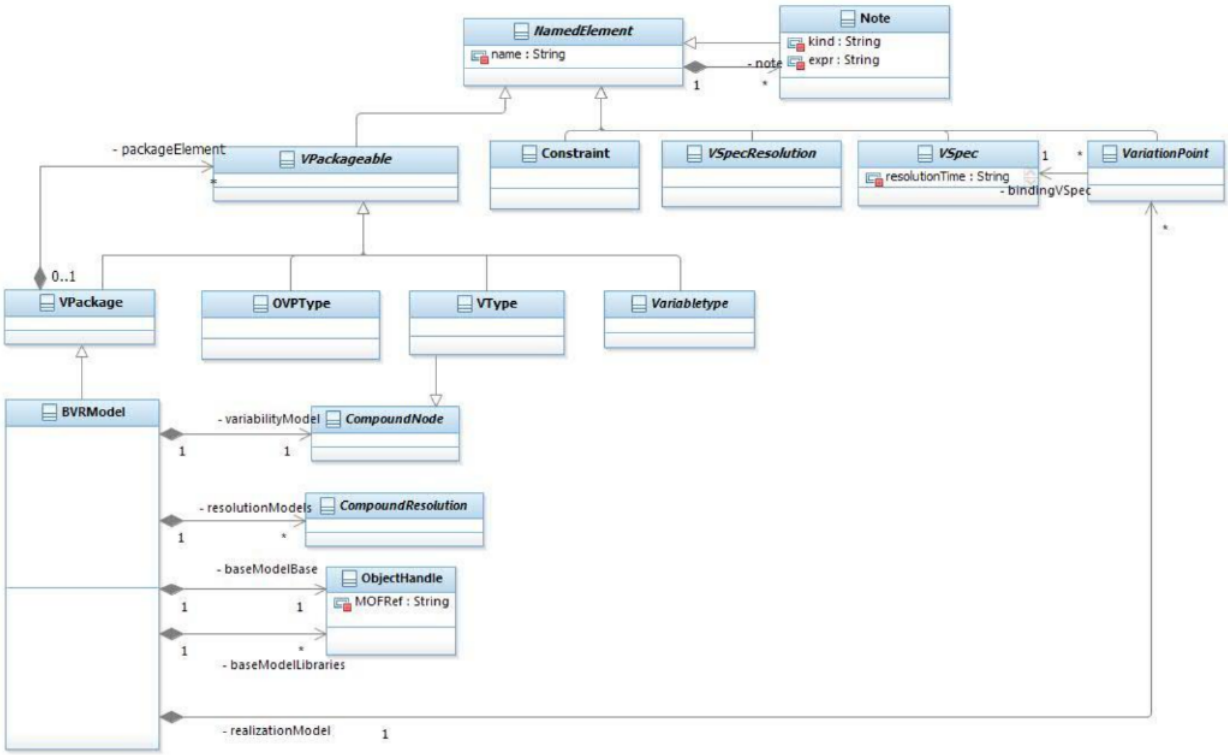

Figure 3.10: BVR Top level

is to support the orthogonal documentation of product line variability, product lines variability is documented in a dedicated model.

Fitzgerald et al. propose in [28] to extend the orthogonal variability model of the product line paradigm in model-based systems engineering and to adapt it to the specific needs of the industry.

This approach is closer to a description of constraints related to SysML system models and orthogonal variability at the same time. They start with a discussion on the different aspects that need to be covered for expressing variability in systems engineering and relate them to a list of contextual requirements for variability management : types of variability, sources of variability, legacy variability specification, constraints, variability in system architectures models, multiple viewpoints, and customer-oriented variability.

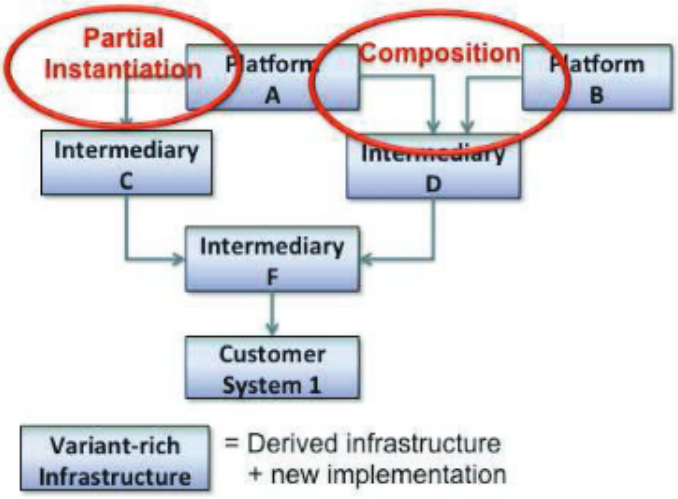

Figure 3.11: Multi-Platform Situation in a Software Ecosystem

3.7 Variability Extraction

Many authors have tried to automate the variability extraction process. We provide in this section a selection of the works done in this specific field.

3.7.1 Variability extraction approaches

Assunçao attempted to extract variability-safe feature model from source code dependencies in [6]. Their main goal is to reverse engineer software using implementation knowledge artifacts. Their approach has other objectives such as standard precision, recall metrics and variability-safety (proving that all the features combinations are well-formed software). This approach is evaluated by five case studies.

Font et al. tried to automate the variability formalization using CVL in [29]. The specification of common parts and differences of the model is done using placements over a base model and replacements in a model library. It is possible to derive new products from the generated software product lines, and even to evolve the SPL by the creation of new models. This approach has been tested on a real case study.

3.7.2 Variability extraction case studies

Galindo studied the Debian repositories as software product line models in [30] (figure 3.16). The Debian repositories indeed seemed like a good case study for a large variability models. They established that the dependency language used for the Debian packages could be interpreted as a software product line variability model. They then provided an automatic analysis of these models, especially to find anomalies.

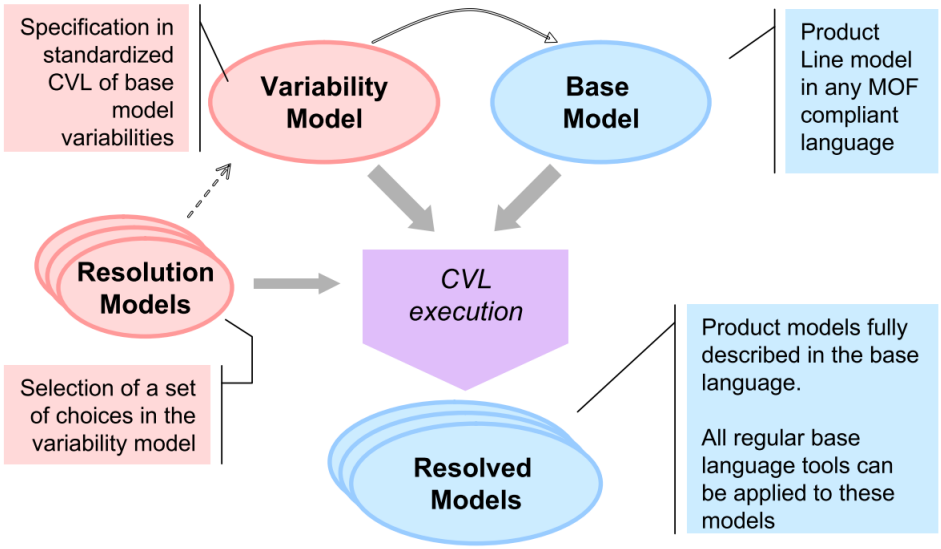

Figure 3.12: Common Variability Language as specified in the Request for Proposals

3.7.3 Variability extraction tools

Ter Beek developed another tool for product variability analysis in [67]. The tool named VMC, starts from the specification of a product family specified, then automatically generates all the family'd valid products. It then checks the model properties and temporal logic.

Zhang and Beker present code-based variability model extraction in [74]. Their tool is aimed at large SPL where the knowledge has been lost. It analyzes thus existing large-scale SPL reuse infrastructure in order to detect improvement potential according to metrics. They also base their research on Conditional Compilation (CC, a variability mechanism based on #ifdef statements in code core assets) and proposes a pre-parsing of variability, followed by automatic extraction of an implementation variability model using a hierarchical variability tree structure. Their approach is then applied to an existing SPL. An interesting point in this study is the use of Goal-Question-Metric (GCM) approach to determine the interesting metrics to attain the goals.

Finally, VARIES is a project about variability as a whole ([32]), defined by its authors as "an industry driven research project in the ARTEMIS program on the topic of variability in safety-critical embedded systems" [1]. This deliverable of the project focuses amongst others on Automatic Support For Commonality And Variability Analysis and establishes a methodology to implement variability in industrial processes.

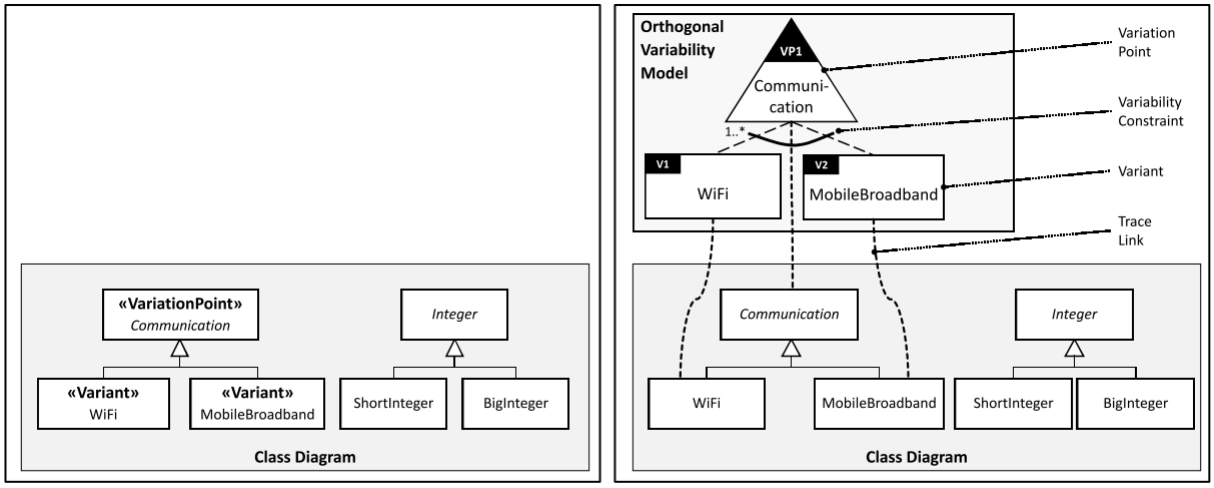

Figure 3.13: Illustration of Two Major Approaches for Explicit Variability modeling, [45]

3.8 Variability Evolution

As we will work on software system that will evolve over time, it is interesting to look at the way variability evolution has been studies until now.

Sbai et al. conceived a pattern based methodology for evolution management in business process reuse in [61]. This methodology manages evolution of configurable process models in terms of activities, data and resources. The process patterns systems are used for an automated support so as to manage the evolution of configurable process models.

Hellebrand et al. analyze coevolution in the industry and look after coevolution in [34]. They look after code artifacts and variability models over some period of time in order to highlight the relationships code and variability model have as they evolve. They also we suggest metrics to detect variability erosion in the code based on changes in the variability model. They use a GQM graph to identify their metrics (see figure 3.18). They finally establish equivalence classes of coevolution (table 3.1)

Niechzial presents a framework named Codeface to visualize and analyze product-line evolution [49]. Codeface has many heterogeneous components (see figure 3.2) and is an example of an integrated variability evolution analysis tool. It includes especially

– Version Control System (VCS Analysis) to extract information about a software project

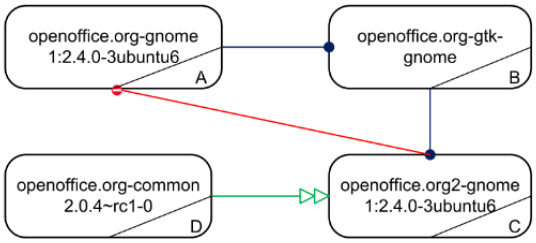

Figure 3.14: Debian VML example

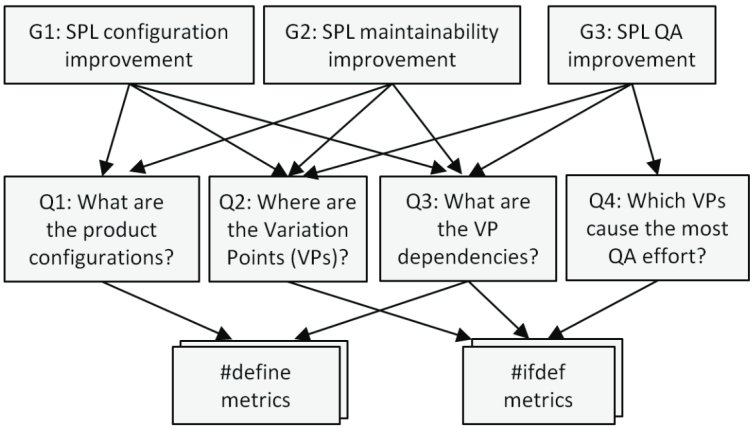

Figure 3.15: The GQM Model of SPL Improvement, [74]

– ML analysis for mailing lists
– A bug extractor that fetches information from bug tracking software about the bug reports

3.8.1 Variability evolution case studies

Passos et al. studies the Linux kernel in ([52]) in order to get a better understanding of coevolution of variability models and related artifacts in "large and complex real-world variability-aware system." They present a catalog of evolution patterns, including the coevolution of the Linux kernel variability model, Makefiles, and C source code. They list the following patterns :

– Add Visible Optional Modular Feature (AVOMF)
– Add Visible Optional Guard Modular Feature (AVOGMF)
– Add Visible Optional Non-Modular Feature (AVONMF)

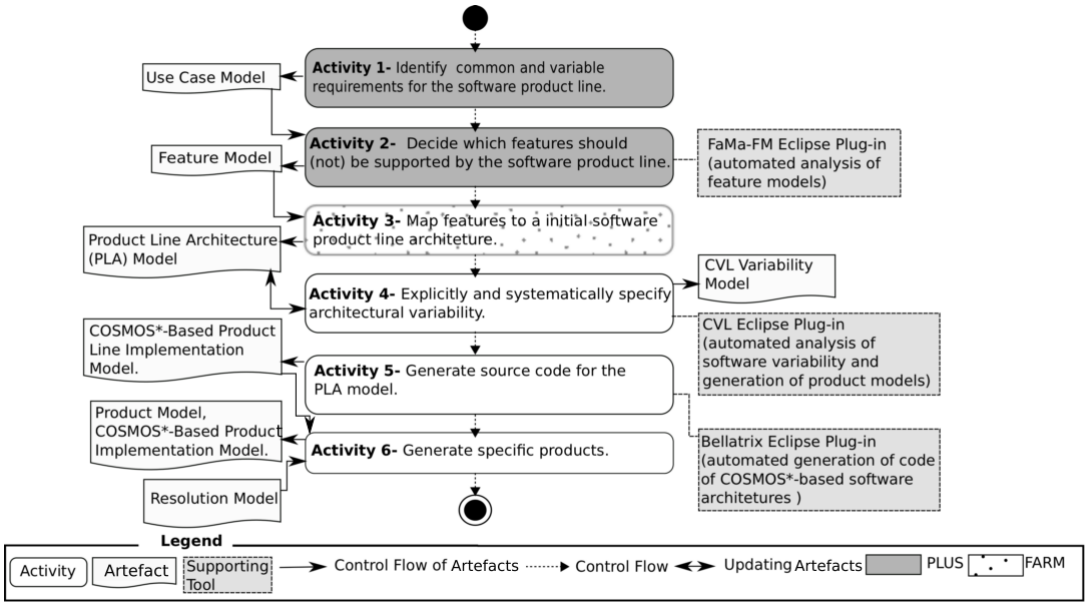

Figure 3.16: A semi-automated model-driven method for developing SPL architectures, [30]

- Guarded directory example (AVOGMF)
- Add Visible Optional Non-Modular Feature
- Add InternalModular Feature (AIMF)
- Featurize Code (FTC)
- Retire Feature (RVOMF, RVOGMF, RVONMF, RIMF)
- Merge Visible Optional Feature into New One (MVOFNO)
- Visible Optional Feature into Computed Internal (MVOFCI)
- MergeVisibleOptional Feature into Sibling (MVOFS)

3.9 Variability Testing

3.9.1 Variability testing approaches

Henard et al. proposes a genetic algorithm to handle multiple conflicting objectives in test generation for SPLs in [35]. They start from feature models and test suites to develop their genetic algorithms (which are "search-based heuristics mimicking the natural evolution process.") in a multi-objective genetic algorithm.

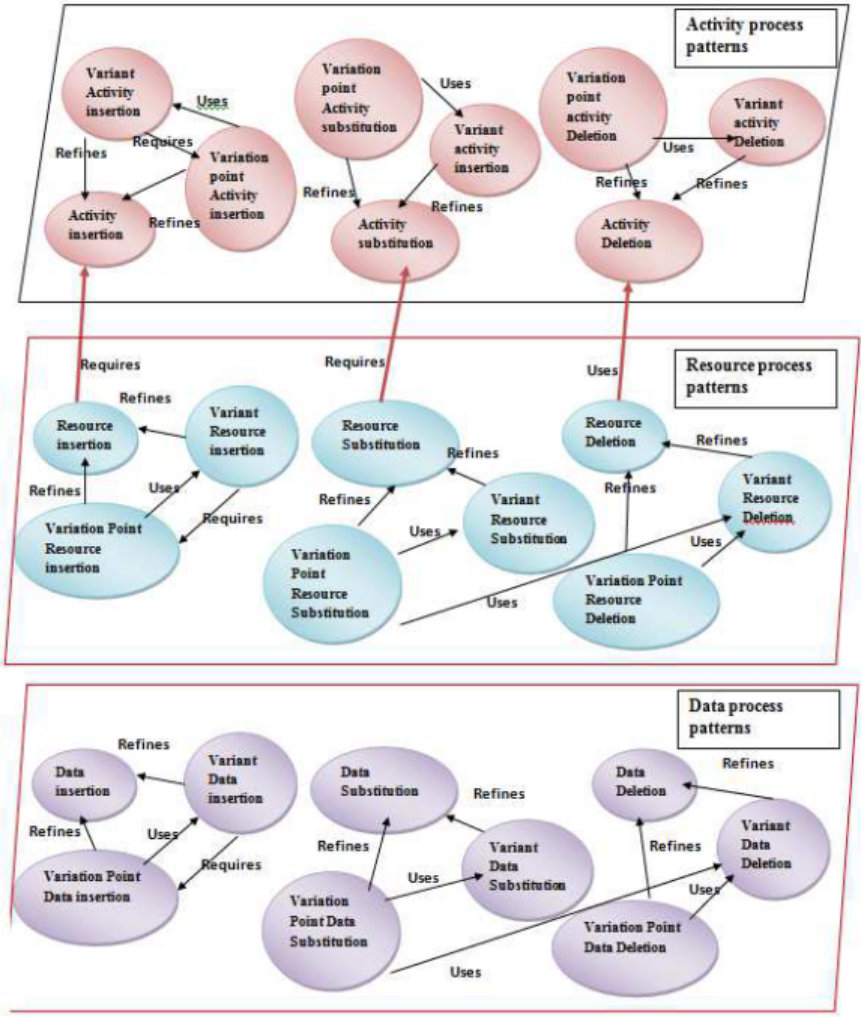

Figure 3.17: The process patterns systems, [61]

Svendsen et al. presents an approach for automatically generating a testing oracle for train stations using the Common Variability Language in [66]. They define the testing oracle by specifying the precondition (test input).

Huhn and Bessling proposes a uniform integration of requirements into a model-driven feature-oriented design methodology of product lines in [36]. Their research is about the case of safety requirements in medical devices, for which the safety requirements vary and the product vary as well. They consider the design modifications and the adaptation of safety requirements as a feature, which are described as a model graph transformation. Their approach uses CVL as the variability modeling language.

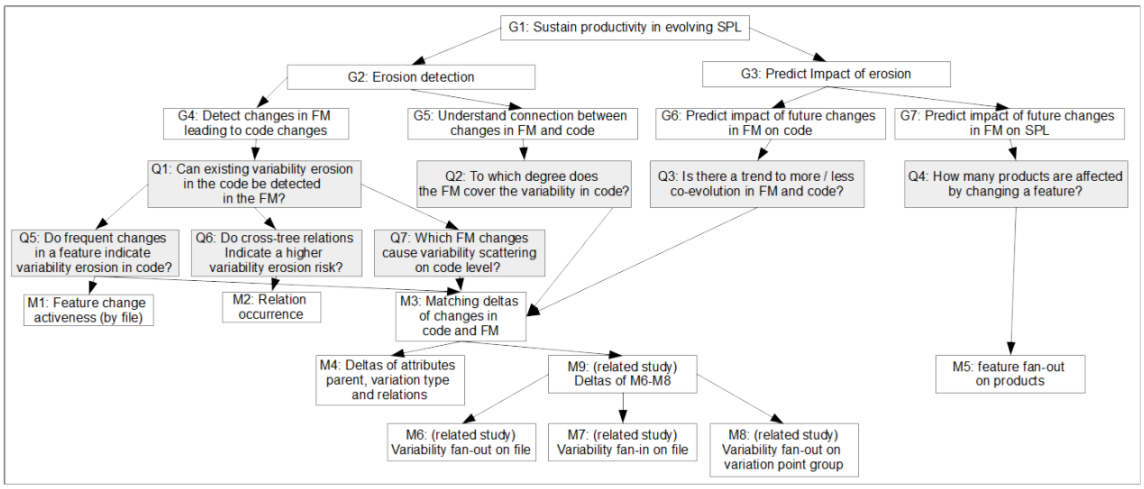

Figure 3.18: GQM Graph, [34]

3.9.2 Variability case studies

Salvador performed a controlled experiment to assess the test effectiveness of using fault models to provide SPL testing with support to design test inputs in [59]. Starting from the hypothesis that "improvements in the quality of variable assets entail addressing testing issues both from high and low-level perspectives", he designed an approach for building fault models for variability testing, with two main perspectives: test assessment (evaluation of the effectiveness of existing test suites) and test design (help for the construction of test sets).

Sánchez et al. uses the Drupal framework as a case study to evaluate variability testing techniques in [60]. They represent the variability of the Drupal framework using a feature model and then report non-functional data. They find positive "positive correlations relating the number of bugs in Drupal features to their size, cyclomatic complexity, number of changes and fault history."

3.9.3 Variability tools

Meinicke et al. provide a first overview on software product line analysis tools in [44]. They establish four characteristics of product-line tools : product-line implementation techniques, software analyses, strategies for product-line analysis and strategy of the tool. They then start from the sampling (generating a representative subset of products), and then move to tools for product-line verification : type-checking, static analysis, software model checking, theorem proving, consistency checking. They also mention

Equivalence Class	Description
E1: synchronous change	Code and feature model delta reflect each other
E2: no change, variability not reflected in model	Due to prior independent evolution of the code, there exists no synchronicity of feature model and code.
E3: no change, feature not reflected in code	Due to prior independent evolution of the feature model, there exists no synchronicity of feature model and code.
E4: independent code evolution	There is a change in the code without a corresponding change in the feature model.
E5: independent model evolution	There is a change in the feature model without a corresponding change in the code.
E6: delayed coevolution of code	A former independent evolution of the feature model is reproduced on the code level.
E7: delayed coevolution of model	A former independent evolution on the code level is reproduced in the feature model.
E8: independent, contrary evolution of code and model	Feature and variability in the code evolve in different directions, i.e., an existing feature of the feature model is deleted; in the same version, the corresponding variability is introduced into the code with a delay.
E9: no change, variability in code and feature synchronized	Feature and corresponding variability in the code are synchronized and are changed neither in the model nor in the code of this version.

Table 3.1: Coevolution Equivalence classes, [34]

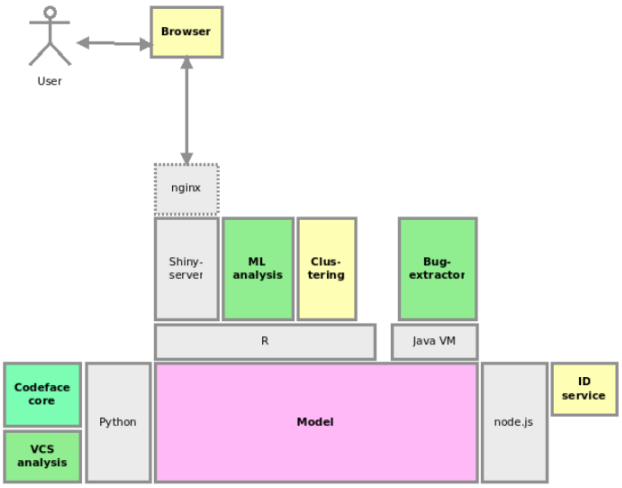

Table 3.2: Codeface architecture overview, [49]

non-functional properties and code metrics. They finally deliver two overviews of the tools according to these criteria.

3.10 Software product line migration

Assunção and Vergilio describes the main outcomes of a systematic mapping study on the evolution and migration of systems to SPL in [7], as it is possible to improve

migration approaches by mapping the works on the subject and the identification of the research gaps. They define three phases for the migration process :

- Detection phase : extract relevant information from the input artifacts
- Analysis phase : design and creation of new partitions that cluster the optional features
- Transformation phase : execution of the transformation on the artifacts

Dillon and Rivera explores the approach to product line adoption across the development organization and successfully adopt second generation product line processes in [17]. They establish a migration between product line technologies as follow :

1. Isolation
2. Awareness
3. Shared Delivery
4. Shared Baseline
5. Collaborative Development

Parra et al. presents an industrial experimentation and a proposal for an SPL adoption in Heinsohn Business Technology (HBT), a software development company specialized in financial, transportation, mortgage-backed securities, and pension-fund solutions in [51]. They state that there are three different levels of variability : business, functional and platform. They see two main challenges for the derivation process for the SPL : first, modeling multi-level variability, and then the multi-level constraints and configuration correctness.

3.11 Databases Variability

There is currently not a lot of work done for implementing variability in database applications.

We can still cite the work of Broneske et al. in [13] which explores approaches for mastering variability in database systems in order to customize them for specific applications. They characterize three approaches (one-size-fits-all, specialization and software product lines) and state that the SPL approach is the best one.

Another one is the work from Khedri who handles database schema variability in software product lines ([41]) thanks to delta oriented programming. Delta programming is an approach where "a product is generated by adding delta modules to a core module incrementally [62]. They compare delta programming and feature-oriented programming and provide a way to consistently check DDL scripts using three rules :

- Existence rule : existence of a database object is assumed
- Non-existence rule : non-existence of a database object is assumed
- Reference rules: it is necessary to check the references to a primary key

Mori and Cleve have developed a generic framework to adapt database schemas according to the context in [47]. The framework considers two levels of abstraction for database schema specification : the conceptual schema and the logical schemas, and maps one to another. It also provides feature-based schema filtering (especially adaptation scenario).

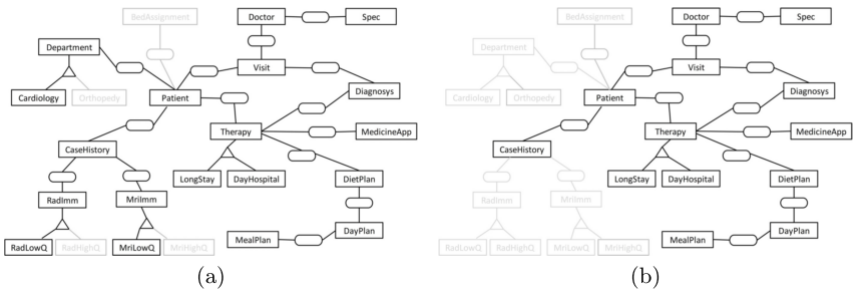

Figure 3.19: Source (a) and target (b) data schema configuration

In [73], Zaid and De Troyer provide an approach for modeling data variability as part of the overall software product line modeling approach. They extend their Feature Modeling technique with the persistency feature concept. They use the EER model to represent the variability. Their approach maps the features of the product line and the variable data model and thus allows the automatic derivation of the actual data model.

Bartholdt et al. describe in [10] the Approach for Data Model Variability (ADMV), based on UML and which is a methodology to provide "a consistent view to capture data variability in data models". ADMV is then applied to a theoretical eHealth software product line.

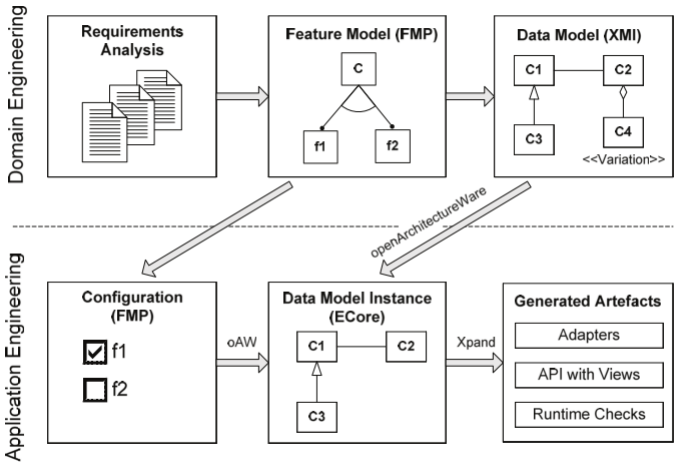

Figure 3.20: ADMV Process

In [50], Nystrom et al. present COMET, a Component-Based Real-Time Database for Automotive Systems (RTDBMS, or real-time database management system) designed to solve the problem of data management variability in different systems. They detail the architecture of COMET (see figure 3.21)and provide an example of use.

Finally, Siegmund et al. tries to bridge the gap between variability in client application and database schema in [64]. Their goal is to generate custom database schemas by applying software product line methodologies to database schemas. Two mains approaches are used in their work. The first of them is the "Virtual decomposition" which consists of leaving all the elements in the same schema but only highlighting the relevant ones. The second approach is the "Physical decomposition" thus storing schema elements in separate files.

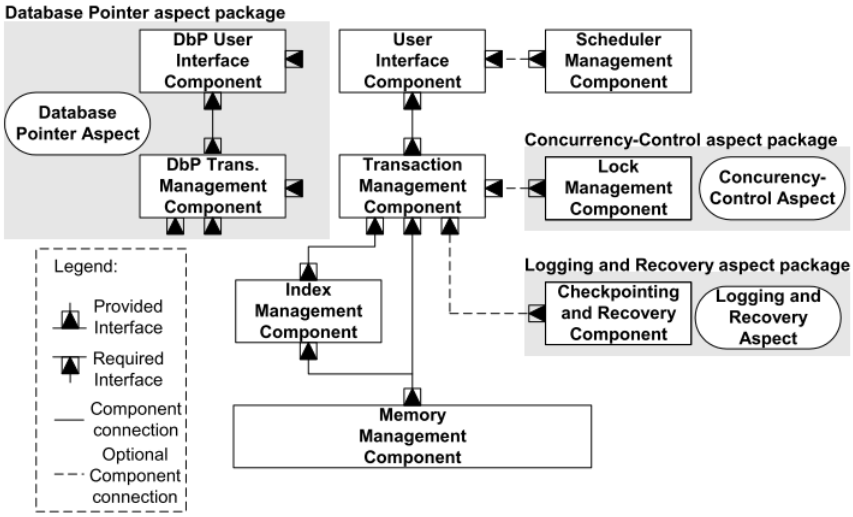

Figure 3.21: COMET architecture

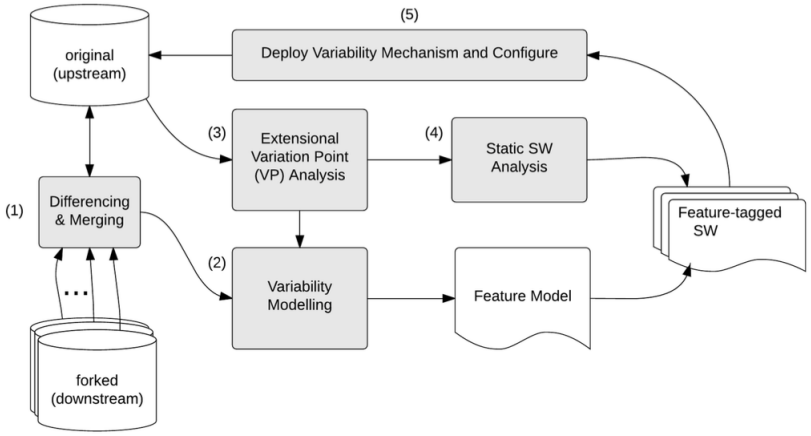

Figure 3.22: Multi-repository to product-line reengineering process: overview

3.12 Relevance of the research question

As we have seen in this state of the art, the question of variability in database has not been fully explored yet. Several techniques exist to manage variability from a model, but this model is rarely a database. Furthermore, it is even a rarer situation that the question of the consistency of the database model is studied.

We present in the following chapter the context of our study: the OSCAR system.

Chapter 4

Case study: A Brief Introduction

In this chapter, we will briefly discuss about our case study that has motivated our research. The goal of this section is to give to the reader a global comprehension of the problematic, then we will present the development of our solution in the following chapters and finally conclude our case study by the application of the tool we developed.

4.1 Context : the OSCAR ecosystem

OSCAR is a fully featured Electronic Medical Records (EMR) software program, designed by doctors for doctors, for use in medical offices. For a quick list of the key program features click here. Besides by physicians, OSCAR is also used by a variety of other front line health care professionals, including registered midwives, social workers, psychologists, nurse practitioners and physiotherapists. OSCAR is an OPEN SOURCE project. To our knowledge OSCAR is the only widely deployed open source EMR system in Canada. The name "OSCAR" is an acronym for "Open Source Clinical Application Resource". [2]

4.1.1 Architecture

OSCAR is built around a client-server model where different clients access a central server over a local network or the Internet. OSCAR is written in server-side Java. It features a web interface written in , JavaScript, Java Servlet and Java Server Pages Script. The default back-end database is MySQL. The default installation of OSCAR runs on the top of the Apache Web server with Java Runtime Environment enabled via the Tomcat application server. Ubuntu Linux is the preferred operating system although OSCAR runs on other distributions of Linux, Windows and MacOS. OSCAR is modular software composed of various modules that implement different functions. [58]

4.1.2 Variability Representation

Figure 4.2 shows an example of feature model of OSCAR, from the work of Weber et al. [70].

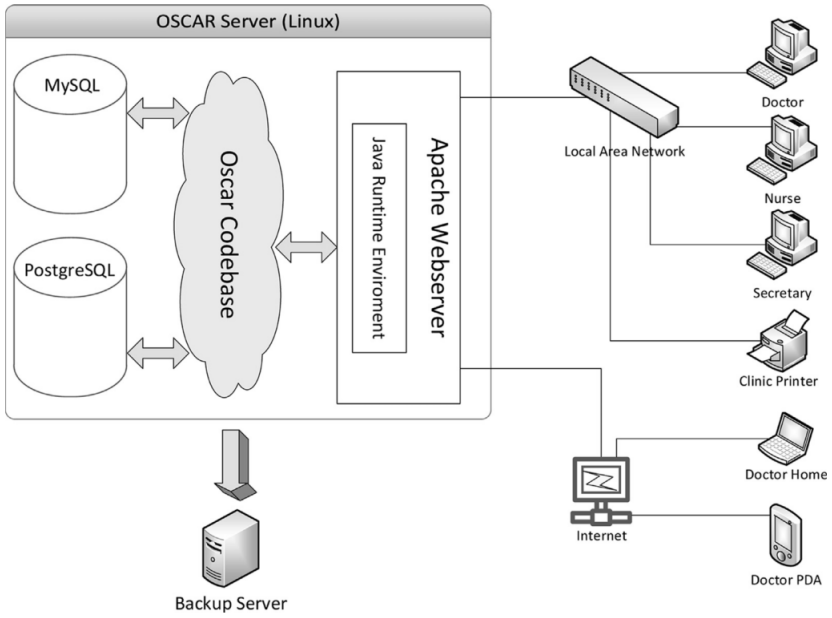

Figure 4.1: A typical Oscar installation, [58]

4.2 Variability Management Issues in OSCAR

Weber et al. present in [70] their research on the application and combination of tools and methods for uncovering and managing variability in the context of real-world industrial case study in the health care domain.

According to them, "OSCAR is customized, deployed and maintained by private, for-profit companies commonly referred to as OSPs (OSCAR Service Providers). While all OSCAR products share a common core code base, significant differences exist between OSCAR products provided by different OSPs. Some of these differences are rooted in differences in deployment context, e.g., different provinces have different healthcare system structures, which result in the need for different billing methods, interfaces for laboratory data, pharmacy integration, electronic health exchange standards, coded data terminology sets, etc. Other differences are due to different customer requirements (e.g., midwife clinic, GP, hospitals) or varying delivery models (e.g., cloud hosted delivery vs. clinic installed system). Moreover, OSCAR software is subject to various quality certification regimes depending on the jurisdiction it is deployed in."

The main issue is thus that "The main OSCAR code repository is based on the distributed SVM Git and hosted in Ontario. OSPs often have their own forks of the code in order to maintain their own consistent variation of the OSCAR product. Given this landscape of different code repository forks and branches, it has become increasingly

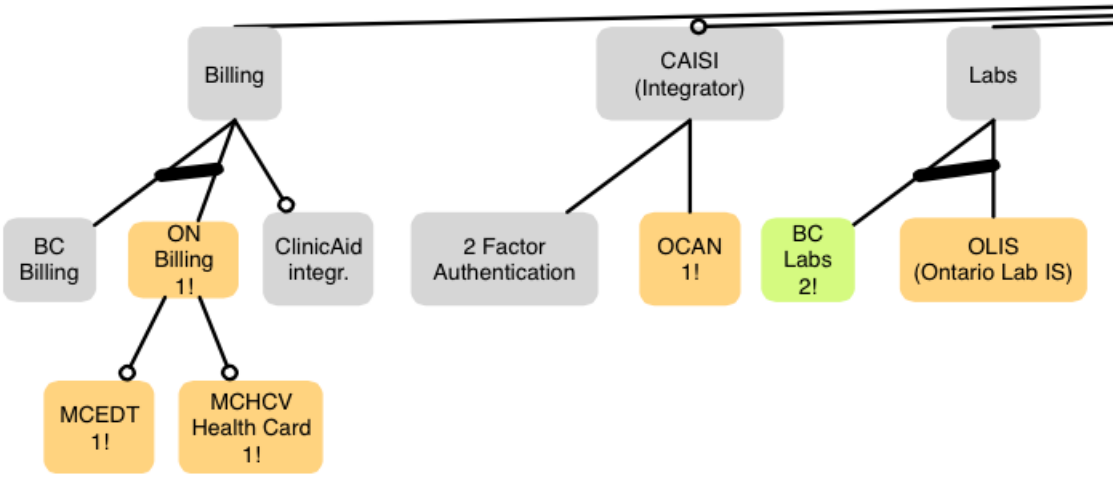

Figure 4.2: An example of feature model for OSCAR

burdensome to effectively manage the variability of the OSCAR product line in presence of continuous updates and development. The Universities of Victoria and British Columbia have partnered with several OSPs in the province on a collaborative research project with the aim on improving variability management practices of the OSCAR software. The goal is to reduce the reliance on different, disparate code repositories for managing different product versions "in space" and migrate to a common explicit variability management process that will ultimately reduce cost and increase quality assurance capabilities."

4.3 Motivation

If we take the CVL example seen in the state of the art, multiple steps are needed in order to develop a variability model. Is every one of them really necessary? With large systems such as OSCAR in mind and the number of variability points, could we not design a tool allowing variability management on large databases without an unnecessary complexity? Indeed, the database reverse engineering process needed in order to manage variability requires enormous efforts (source needed) It would so be interesting to have a tool reducing the workload of this task.

We present in figure 4.3 steps seemed necessary and sufficient which should be performed using a new tool to help us in the management of the variability in our case study. The first step should be the understanding of the domain and extraction of variability. This step, however, is not part of our scope. We are not experts and OSCAR our goal is not to develop a tool to facilitate the extraction of variability. So we focus on 4 stages.

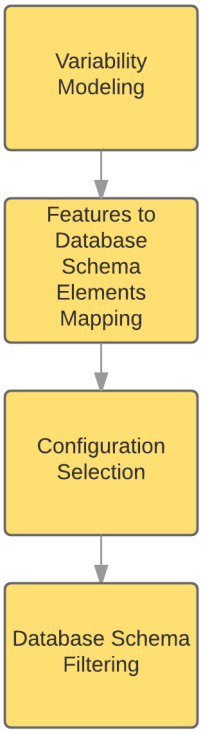

Figure 4.3: Steps required in the variability management tool

For our case study, assuming that we have the necessary knowledge of OSCAR obtained following a domain analysis and extraction of variability phase ,we would like a tool to quickly produce a representation of our model, then map this model with the database schema elements, select a configuration in our model and finally filter our database accordingly with the configuration selected.

In the following chapters, we will explain the developement of our tool and this case study will be continued in chapter 8.

Chapter 5

Methodology

In this chapter we present the methodology followed to meet our goals defined in the previous pages, but also the problems encountered and the techniques we developed to face those problems.

5.1 Methodology Objectives

As we stated in chapter 4, the main objective of our research was to find the best way to deal with the variability in the context of database applications, more precisely we had to develop a tool to represent and to manage the variability present in the applications, especially at the database level. Since the beginning of our work, it became quickly clear that there were three main goals in our work, everyone of them being a step in a more global process,

Quite naturally, our first goal is to **represent variability**, in order to express the domain particularities into a variability model.

Starting from this variability model, our second goal is to **map the features to the database schema**.

Finally, our last objective is to **adapt a schema based on variability choices**. We thus need a step that allows the user to **select a configuration** for the future system.

Figure 5.1 illustrated this first glance at our methodology.

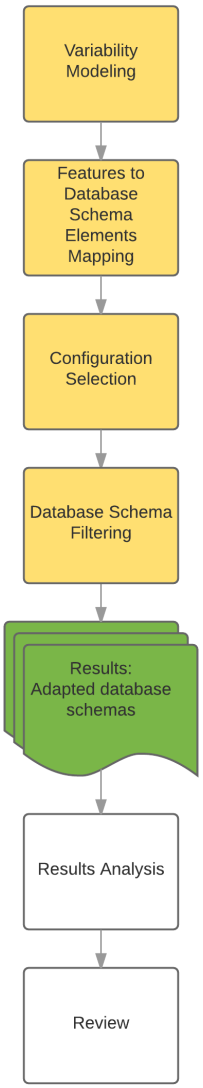

Figure 5.1: A high-level view of our methodology

5.2 Variability Modeling Methodology

We explain here our approach to find the variability modeling language that would suit our needs. The first step of our methodology, as represented on Figure 5.2, was to elaborate the language that would suit our needs. The variability requirements are the starting point, as we needed to know exactly what constraints we had to comply with in order to answer the question. After having determined all the requirements, we started to compare the existing variability modeling languages. Then comes the Variability Modeling Language Choice that is a process we describe later and which gave us the Variability Modeling Language appropriate for our situation.

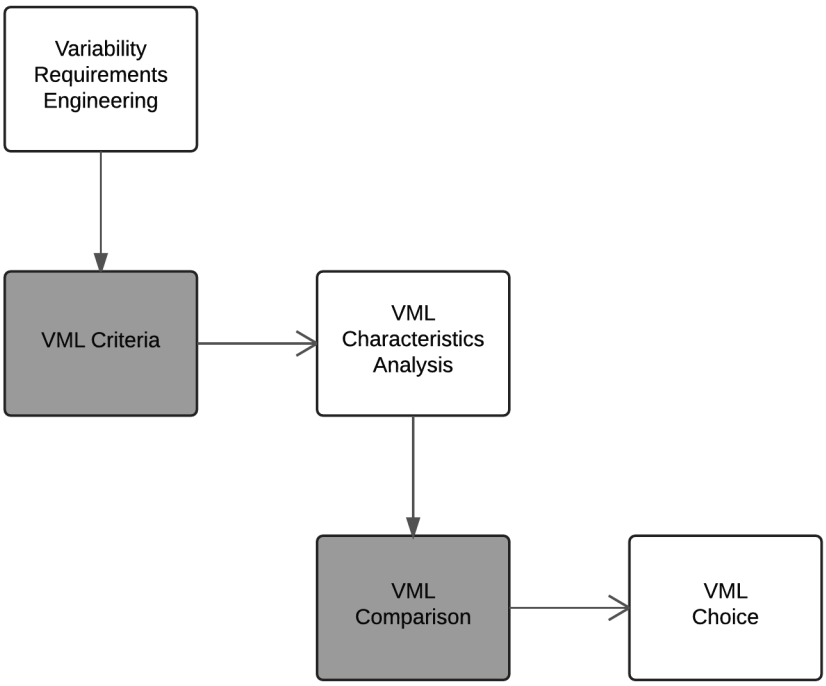

Figure 5.2: Our Variability Representation methodology

Variability Requirements Engineering

The specification of the variability modeling language was the first phase of the process, and the more creative one. Starting from the research problem, we had to imagine all the features we needed in our variability modeling. We used methods such as brainstorming, use cases and functional requirements to establish our vision for the best variability modeling language for us.

Here are the following characteristics we wanted for our variability modeling language.

First of all, the **scalability** was an important characteristic for our language, as it would possibly be used on the whole OSCAR system and its huge number of tables. We also wanted a **text-based language**, so it could be easier create other tools based on this text structure.The third and final requirement was the respect of **standards**, so that the artefacts produced by the system could easily be handled by standards operations and libraries.

We also established a list of weaknesses that we did not want our language to have. First, the language should not be under development, as it would mean that the language may still be immature. Second, the project language should not be deprecated. Third, it should not suffer from a lack of documentation, and last, it should not be a language with none or with only a few case studies.

Variability Modeling Language Characteristics Analysis

Based on the variability modeling language criteria established in the previous step, we produce a comparison of the existing variability modeling languages. This comparison is a table with four columns : first, the VML name, then the VML neutral characteristics, the VML positive characteristics (also known as his "strengths") and finally the VML negative characteristics (also known as his "weaknesses")

The languages we considered to model variability are presented in section 3.5. Table 5.2 is a comparison between every of them to explain our choice.

Variability Modeling Language Choice

IVML was a promising variability modeling language, with good conceptual documentation. However, the tool it is used with (Easy-Producer) was still under development, and the technical documentation was considered not detailed enough yet.

We considered BVR, a language based on CVL. Unfortunately, this project was still under development and lacked maturity.

We also researched CVL. As CVL had already been used in Java Code transformations (as seen in [27]), it seemed like a good language for what we wanted to do. We became familiar with the CVLTool Eclipse plugin and decided to chose this modeling language over the others. CVL however was too complex for what we needed. Its meta-model includes substitution fragments, replacement fragments, placement fragments. All these structures are unnecessary for our needs. This is why we have decided to design our own variability modeling language, based on the general philosophy of CVL.

	Characteristics	Strengths	Weaknesses
Base Variability Resolution	– based on CVL	/	– still under development
Integrated Variability modeling Language	/	– is scalable according to [22]	– too few papers and case studies – the user guide documentation is not intuitive and not complete – the development is still going
Text-based Variability Language	/	– text-based – scalable according to the authors	– not as scalable as other languages according to [22]
Common Variability Language	/	– domain-independent – respects standards	– not developed anymore

Table 5.1: Variability modeling Languages Comparison

Constraints

Our methodology for the constraints was the following : brainstorming on the constraints that needed to be used in our model, and then what rules had to be set to ensure their consistency.

5.3 Mapping Features to Database Schema Elements

The variability models created with to the Variability modeling Language need to be linked to the elements of the database schema. We thus had to first study the database schema elements, then the definition of the feature elements in order to get a possible mapping between them that we could combine with our variability modeling tool in order to design a complete working tool.

As far as the features elements are concerned, they are defined in the variability representation section.

The first step to create this mapping is to choose the type of database schema we want to work with: physical, logical, conceptual?

The second step of the mapping process is to define the precise mapping between the features elements and the database schema elements. This issue is developed further in chapter 6.

5.4 Configuration Selection

This step represents a well known notion in software product lines: the configuration. It is the link between the mapping of the base domain to the variability model and the filtering phase.

5.4.1 Resolution Model and Configuration

We decided to define two concepts in this part of the methodology. We start with the assumption that the mapping phase produces an artefact, which is basically a variability model improved a mapping and some constraints. We call this artefact a **base model mapped variability model**. It is still generic, and represents all the possible choices for a defined variability model.

If a base model mapped variability model represents all the possibilities, it is possible to instantiate it into a particular version of the variability model we are considering. This process is possible through choices, and result in the production of a fixed variability model, which we call a **configuration**, or a **resolution model**.

5.5 Database Schema Filtering

This last process consists of adapting the existing database schema to the new requirements of the variability model. Concretely, this means that the new database schema will only contain the database elements that are relevant for the choice made on the variability model.

The issues of this part of process are the following: first, to choose in which way the database schema will be modified, and then in a second time, to design the algorithm to ensure this adaptation.

5.5.1 Filtering Strategy

On our way to create a new Variability modeling Language, we went through a critical question : would it be more adequate remove useless tables to create the new database schema (subtractive) or to construct the new database schema from scratch (constructive).

Subtractive approach

In a subtractive approach, the starting point to create a variability model can be any subset of tables we want to study. The paradigm to build the new database schema is the following : if a table is **included**, then it is in the new schema, if a table is **excluded**, then it is not in the new schema, if a table has **no state**, then it is in the new schema.

The last point means that by default, all the tables from this subset are in the new schema. The "included" state might thus seem redundant. However it is useful, mostly to analyze rules, and also for the user comprehension.

The major strength of this approach is that it allows the user to work only on a subset of tables at a time.

Additive approach

In an additive approach, the starting point to create a variability model is the whole set of tables of the system. The paradigm to build the new database schema is the following : if a table is **included**, then it is in the new schema, if a table is **excluded**, then it is not in the new schema, if a table has no state, then it is not in the new schema.

The last point means that by default, none of the tables from this subset is in the new schema. The "excluded" state might thus seem redundant. However it is useful, mostly to analyze rules, and also for the user comprehension.

The major strength of this approach is to offer a whole overview of the system, when its weaknesses are that the variability model must be complete for the whole system in order to obtain a new schema and that the complete variability model considers all the tables of the system

Chosen approach

Finally, we opted for a variant of the subtractive approach: the whole database schema is loaded. From there, a new schema is built. Finally, the old schema is filtered by comparison with the new schema. We decided that the process of discovering and documenting variability needed human involvement (semi-automated). A user would specify a feature model and either adds database structures to features or remove database structures (that do not apply) from features. The additive approach was considered too costly and we therefore decided to implement the subtractive approach.

5.6 Global Methodology

In this section we present our global methodology through two scopes: the first one is the in-detail methodology, for a step-by-step procedure when the second is a more global view. We present both as they should be used in different contexts.

5.6.1 In-Detail Methodology

This view of our methodology summarizes all the steps and issues that have to be dealt with in order to implement variability in a database application. This methodology should be used if a software engineer faces a problem that may seem similar to the one we worked on, but differs by some aspects. For example, he may need to work on the conceptual level of the database, or he may use another variability modeling language. In this case, he should follow the in-detail methodology that is illustrated at the Figure 5.3. It is, of course, possible that some of his decisions coincide with ours, in which case he may skip some steps.

5.6.2 Ready-to-Use Methodology

The ready-to-use view of our methodology is for another use case, which may be described as more practical, where the previous paragraph would fit more for a research question. In this case, the software engineer is in charge of transforming a database application into a software product line. He has the database schema at his disposal, and feels like the language and the tool we provide (respectively in chapters 6 and 7) are suited to his requirements. In this case, then he may look at our ready-to-use methodology, which focuses on the most critical steps of the variability management process.

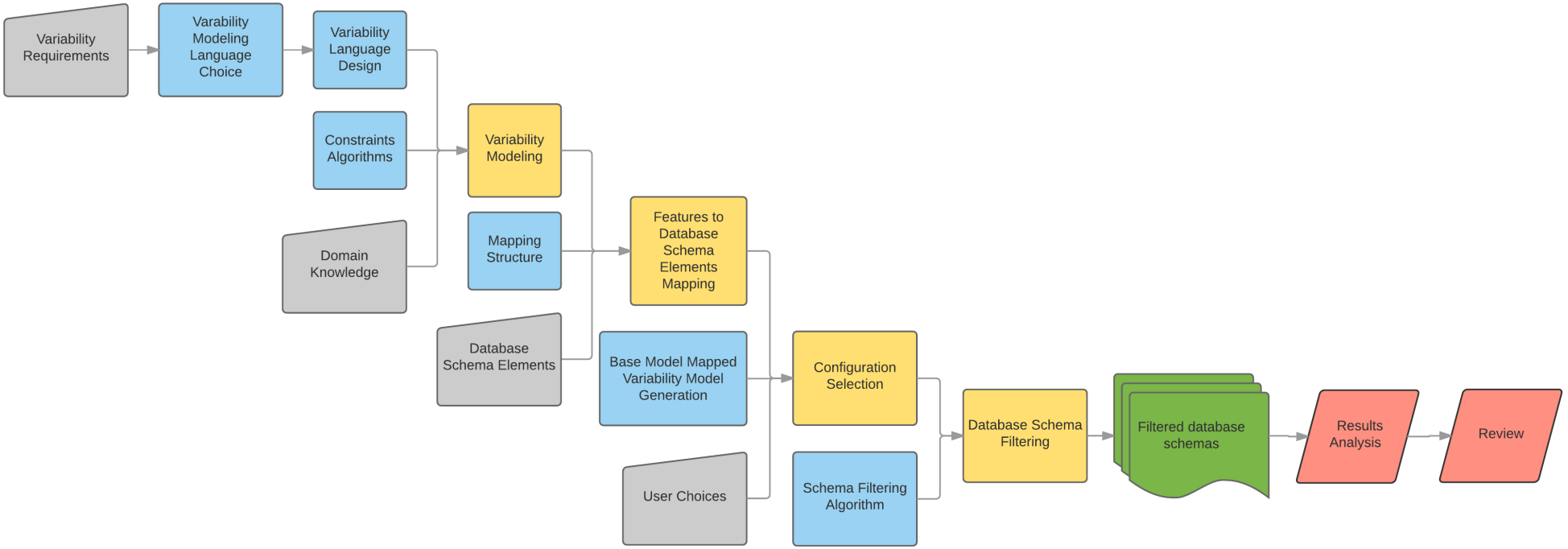

Figure 5.3: In-detail Methodology

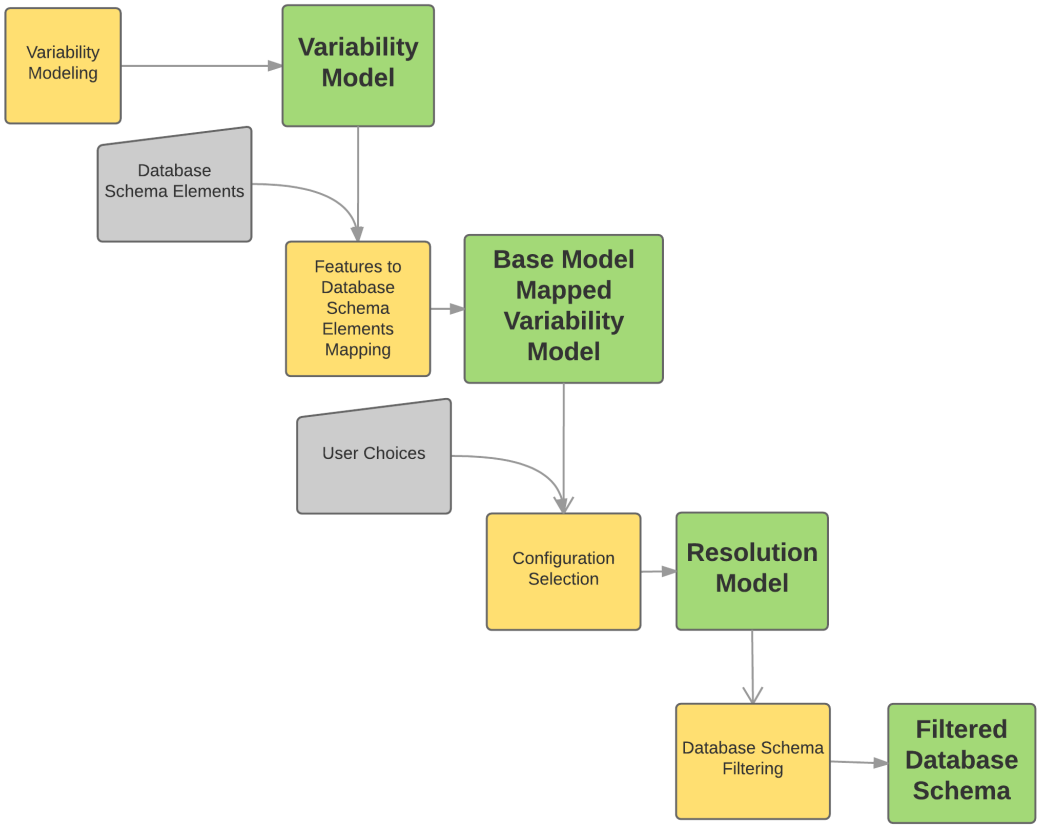

Figure 5.4: Ready-to-use methodology

Chapter 6

Design

After defining our methodology in chapter 5, we specify in this chapter the different techniques exposed in the previous section. We describe here the data structures, algorithms, patterns, etc.

6.1 Global overview of the tool

6.2 Variability Representation Simple Variability Language Design

We explain here how we designed our own variability modeling language.

We started the design of our new Variability Modeling Language with the basis provided by the Common Variability Language documentation and implementation. We thus kept the fundamental elements of CVL, but set aside everything that did not seem relevant for our particular context.

The Simple Variability Language provides constructions to model variability in feature models. A SVL model follows a tree structure and contains the following expressions: Variation Points, Variants, Connectors (OR and XOR), Commons and the Core. The Variation Points are the elements representing the features. The Variants represent the elements needed for a feature. The Connectors express logical constraints between Variants and Variation Points but also between the Core and Variants or Variation Points. The Common child of a Variation Points regroups all the base model elements needed by the potential children of the Variation Points. The Core regroups all the base model elements not included elsewhere.

Constraints may exist between Variant Points and Variants which are not directly related. These constraints are of two types : "requires" and "excludes".

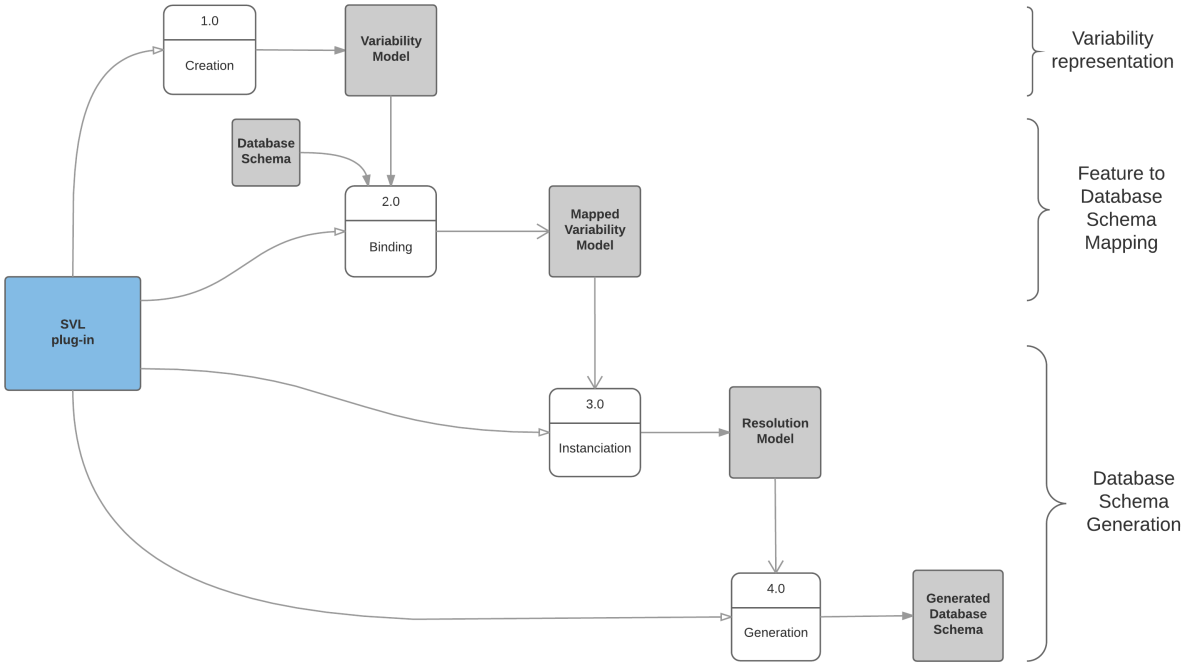

Figure 6.1: Global overview of how the SVL plug-in works

6.2.1 SVL specification

Base Model

- A column can be included in only one variant
- A table is included is a variant as soon as one of its columns is included in the variant.

SVL

- The root of the feature model tree is the Core
- The Core has a Connector child
- The children of a Connector are Variant Points or Variants
- A Variation Point may have a Common child
- A Variation Point has a Connector child
- A Variant references elements of the base model
- Variation Points and Variants can be optional or mandatory
- A Variant is a leaf of the variability model. It can have no children.

Figure 6.2 shows a meta-model of our variability model tree.

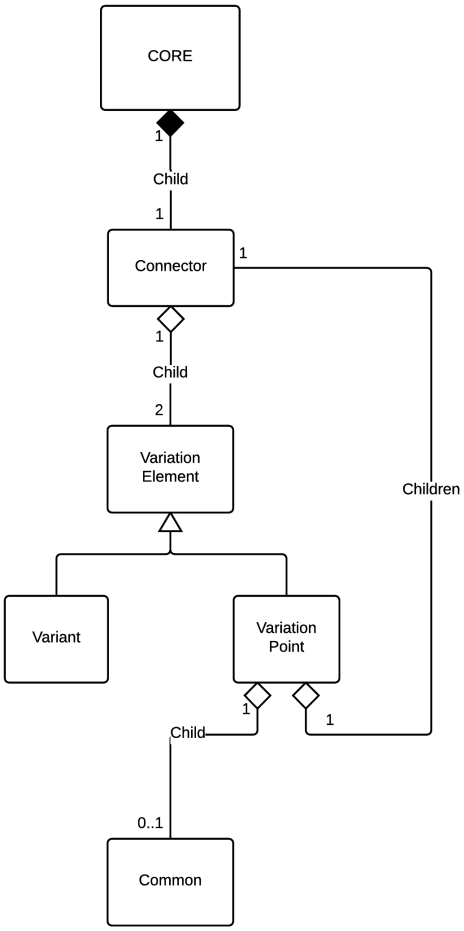

Figure 6.2: Variability Model Tree Meta Model

6.2.2 SVL rules

We distinct two types of rules : the rules and the guidelines. A broken model is a model where an element cannot be selected. The difference between them is based on the consequences for them not being respected : if rules are not respected, then the model is broken, while if guidelines are not respected, the model is still coherent, but contains redundant structures.

Rules

This are the rules that must be respected for a SVL model to be valid.

Requires:

These rules formalize the requirement dependency between variability elements.

1. XOR children can't require each other
2. An element cannot require an ancestor
3. An element cannot require a descendant
4. An element requiring a XOR child excludes the other children

Excludes:

These rules formalize the exclusion dependency between variability elements.

1. XOR children exclude each other by default
2. An element cannot exclude an ancestor
3. An element cannot exclude a descendant
4. An element requiring a XOR child excludes the other children
5. An element exclude mandatory brothers
6. The exclusion of a mandatory element from another branch excludes this last element's father (can be applied multiple times, but must be forbidden if leads to the exclusion of a common ancestor between the two starting elements)

Mandatory:

These rules apply for the mandatory character of an element.

1. The exclusion of a mandatory element from another branch excludes this last element's father (can be applied multiple times, but must be forbidden if leads to the exclusion of a common ancestor between the two starting elements)
2. XOR children cannot become mandatory

Guidelines

These are the recommendations we have for the good practices of SVL. While not mandatory, they often address redundancies or inexplicit issues.

1. If all OR children exclude each other, it should become a XOR
2. If a mandatory element requires another optional element, this one should become mandatory
3. An element should not require its mandatory brothers
4. A mandatory element should not exclude its brothers or any of its brother's descendants.
5. If an element requires its brother, it should be merged with it

Example

An example of variability model created with SVL is shown at figure 6.4 and is based on a small database schema presented at figure 6.3. This schema is used to illustrate the process of creation of a variability model. The variability of this schema is decomposed as follows: Tables such as BC_Payement and BC_Billing are used for the billing in British Columbia while Ontario_Payement and Ontario_Billing are for the billing in Ontario. The schema contains two types of forms, BC forms and eye forms. We also have two types of surveys with Survey_1, Surveys_2 and Surveys_3, Surveys_4.

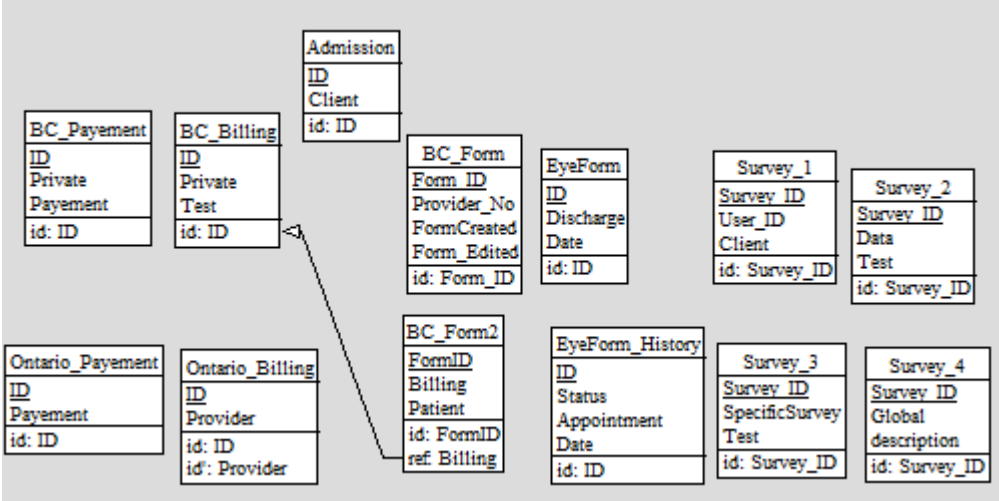

Figure 6.3: Database schema example

6.3 SVL Constraints

We detail here constraints violation patterns that can be encountered. We define in this section the patterns, and for each one, we provide an example with the algorithm we wrote to ensure its checking.

Figure 6.5 shows labels used in the following schemas. For clarity reasons, the other variability models do not contain the core elements, and should thus be seen as parts of a larger variability model. Furthermore, the algorithms that are presented in the following pages are only the high-level processes. For a more detailed view, the other algorithms and procedures are listed in the appendix A.

6.3.1 Base requires and excludes

This type of constraint focuses on the requirements and exclusion between variants. An example may be that Variant 1 requires Variant 2, while Variant 2 excludes Variant 1.

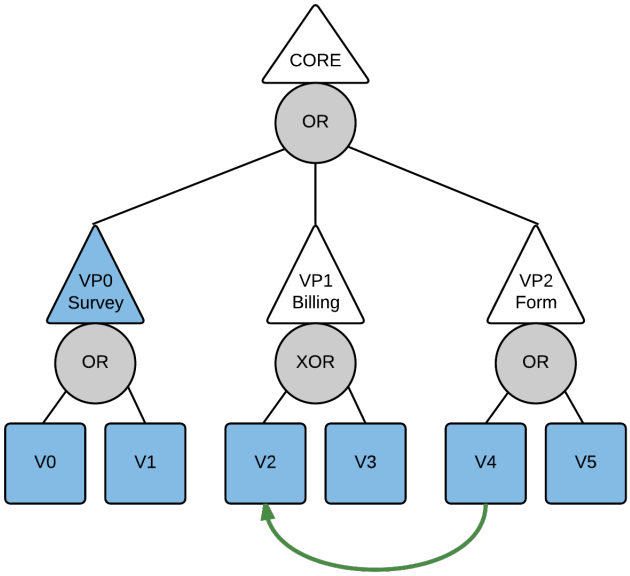

Figure 6.4: Variability model example with SVL

Figure 6.5: Labels

To generate the list of potential elements that a variant or variation point can require, we use a simple algorithm that will first select the elements not already excluded by the current element and then check if the selected ones do not exclude the current element. For the generation of a list of potential elements to exclude, we use an algorithm that will select the elements not required and then check if the selected ones do not require the current element.

6.3.2 XOR Require : XOR children cannot require each other

In this case, Variant 1 and Variant 2 are children of a XOR. Variant 1 cannot thus require Variant 2 (see figure 6.7). The following algorithm (1) ensures this.

Another case is if any other variant requires children of a XOR (see figure 6.8). Algorithm 2 ensures this.

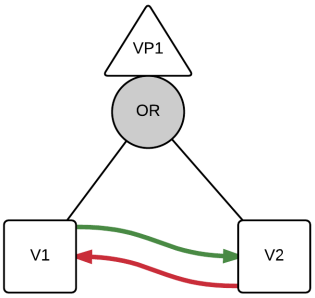

Figure 6.6: Requires and Excludes

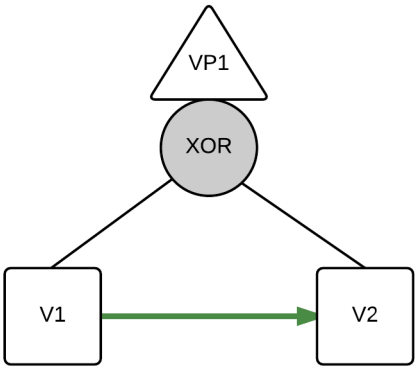

Figure 6.7: XOR child

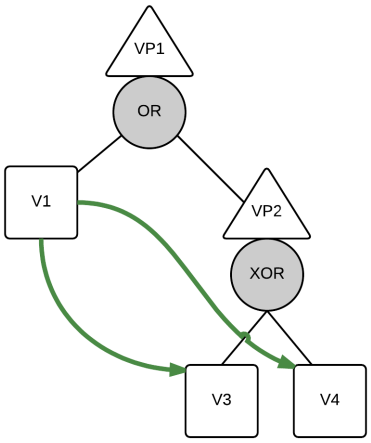

Figure 6.8: XOR children

Algorithm 1 checkXORChildren(root)

```
1: list = root.getListComponents();
2: if (list ≠ null) then
3:     if (root instanceof XOR) then
4:         for (comp0 : list) do
5:             for (comp1 : list) do
6:                 if (IsMutualRequired(comp0, comp1)) then
7:                     error = XOR_REQUIRES;
8:                     listErrors.add(error);
9:                 end if
10:            end for
11:            checkXORChildren(comp0);
12:        end for
13:    else
14:        for (comp : list) do
15:            checkXORChildren(comp);
16:        end for
17:    end if
18: end if
```

Algorithm 2 checkXORChild

Data : Variability Model Results : Rule 2 is checked for the variability model

```
1:  listReq ← null
2:  listRefElement ← null
3:  if (root instanceof Variant||root instanceof VariationPoint) then
4:      if root instanceof Variant then
5:          listRefElement ← root.getReferencedTables()
6:          listReq ← root.getRequire().getReqList()
7:      end if
8:      if root instanceof VariationPoint then
9:          common ← root.getCommon()
10:         if common ≠ null then
11:             listRefElement ← common.getReferencedTables());
12:             listReq ← root.getRequire().getReqList()
13:         end if
14:         listAllRequires ← null()
15:         if listReq ≠ null then
16:             for (Strings : listReq) do
17:                 listAllRequires.add(s)
18:             end for
19:         end if
20:         if (listRefElement ≠null) then
21:             for (c : listRefElement) do
22:                 if (¬listAllRequires.contains(c.getName())) then
23:                     listAllRequires.add(c.getName())
24:                 end if
25:             end for
26:         end if
```

```
27:         if (listAllRequires.size() > 0) then
28:             List < XOR > listXOR = newArrayList < XOR > ();
29:             for (s : listAllRequires) do
30:                 Parent ← Util.getParentComponent(this, Util.getVVP(this, s))
31:                 if Parent instanceof XOR then
32:                     if (listXOR.contains(Parent)) then
33:                         if (listRefElement.contains(Parent) then
34:                             error ← REQUIRE_XOR_CHILD_FK;
35:                             listErrors.add(error)
36:                         else
37:                             error ← REQUIRE_XOR_CHILD;
38:                             listErrors.add(error)
39:                         end if
40:                     else
41:                         listXOR.add(Parent);
42:                     end if
43:                 end if
44:             end for
45:         end if
46:     end if
47: end if
48: List < Component > list ← root.getListComponents();
49: if (list ≠ null) then
50:     for (c : list) do
51:         checkRequireXORChild(c);
52:     end for
53: end if
54:
```

6.3.3 Exclusion of a mandatory brother

Variant 1 and Variant 2 are children of a OR, Variant 2 is mandatory : Variant 1 cannot exclude Variant 2

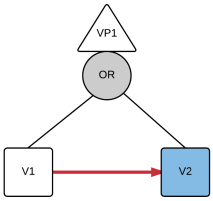

Figure 6.9: Exclusion of a mandatory brother

Algorithm 3 checkMandatoryBrothers

Data : Variability Model Results : Rule 3 is checked for the variability model

```
1:  list ← root.getListComponents()
2:  if (list ≠ null) then
3:      if (rootinstanceof OR) then
4:          for (comp0 : list) do
5:              for (comp1 : list) do
6:                  optional ← false
7:                  if (comp1 instanceof Variant) then
8:                      optional ← comp1.isOptional()
9:                  end if
10:                 if (comp1 instanceof VariationPoint) then
11:                     optional = comp1.isOptional()
12:                 end if
13:                 if (¬optional) then
14:                     if (Util.isExcluded(comp1, comp0)) then
15:                         error ← MANDATORY_BROTHERS
16:                         listErrors.add(error)
17:                     end if
18:                 end if
19:             end for
20:             checkMandatoryBrothers(comp0);
21:         end for
22:     else
23:         for (Componentcomp : list) do checkMandatoryBrothers(comp);
24:         end for
25:     end if
26: end if
```

6.3.4 Transitivity

Variant 1 requires Variant 2, Variant 2 requires Variant 3, Variant 1 excludes Variant 3.

Particular case

Combination of transitivity and mandatory.

6.3.5 Circuit

Variant 1 requires Variant 2, Variant 2 requires Variant 3, Variant 3 excludes Variant 1.

Particular case

Combination of a circuit and a mandatory child.

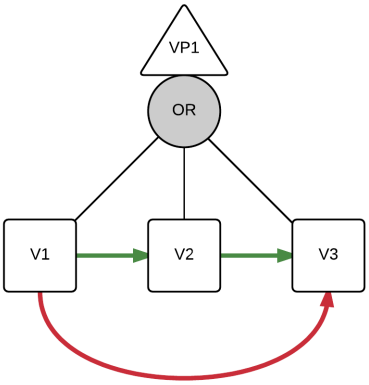

Figure 6.10: Transitivity

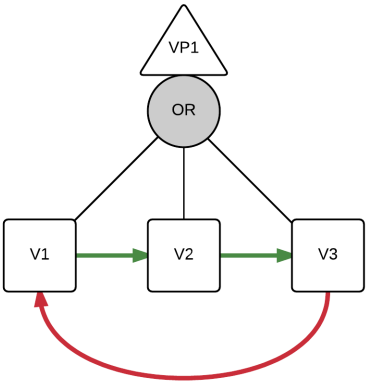

Figure 6.11: Circuit

6.3.6 Crossed Circle

More complicated case of circuit.

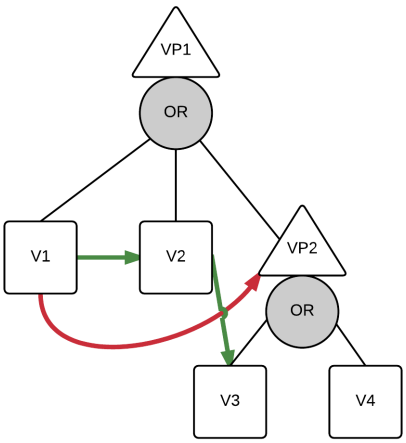

Figure 6.12: Crossed Circle

Algorithm 4 checkCircuitTransitivity

Data :
Result :

```
1:  List < Component > list ← root.getListComponents();
2:  if (list ≠ null) then //TODO modif root instanceof XOR
3:      if (root instanceof OR||root instanceof XOR) then
4:          for (comp : list) do
5:              subCheckTransitivity(comp, new ArrayList, new ArrayList);
6:              subCheckCircuit(comp, new ArrayList, new ArrayList);
7:              checkCircuitTransitivity(comp);
8:          end for
9:      else
10:         for (comp : list) do
11:             checkCircuitTransitivity(comp);
12:         end for
13:     end if
14: end if
```

6.3.7 Contradiction Excludes/Requires and Mandatory

Excludes and requires constraints in conflict.

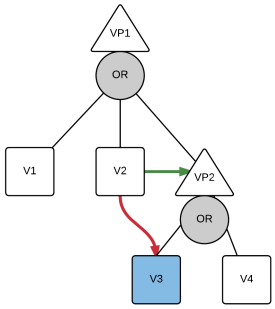

Figure 6.13: Exclusion of a mandatory element

Algorithm 5 checkExclusionMandatoryElement

Data :
Result :

1: **if** (*root* instanceof Variant||*root* instanceof*VariationPoint*) **then**
2: $listRefElement \leftarrow null$
3: $listReq \leftarrow null$
4: $listExc \leftarrow null$
5: **if** (*root* instanceof Variant) **then**
6: $listRefElement \leftarrow root.getListReferencedTables())$
7: $listReq \leftarrow root.getRequire().getReqList()$
8: $listExc \leftarrow root.getExclude().getExclList()$
9: **end if**
10: **if** (*root* instanceof VariationPoint) **then**
11: $common \leftarrow (root.getCommon()$
12: **if** $(common \neq null)$ **then**
13: $listRefElement \leftarrow c.getListReferencedTables());$
14: **end if**
15: $listReq \leftarrow root.getRequire().getReqList()$
16: $listExc \leftarrow root.getExclude().getExclList()$
17: **end if**

```
18:     if (listReq ≠ null) then
19:         if (listExc ≠ null) then
20:             listExcWithMandatory ← newArrayList
21:             for (s : listExc) do
22:                 addExcludes(U.getVVP(this, s), listExcWithMandatory)
23:             end for
24:             for (s : listReq) do
25:                 if (listExcWithMandatory.contains(s)) then
26:                     error ← "EXCLUSION_REQUIREMENT"
27:                 end if
28:             end for
29:         end if
30:     end if
31:     if (listRefElement ≠ null) then
32:         if (listExc ≠ null) then
33:             listExcWithMandatory ← newArrayList;
34:             for (s : listExc) do
35:                 addExcludes(Util.getVVP(this, s), listExcWithMandatory);
36:             end for
37:             for (s : listExcWithMandatory) do
38:             end for
39:             for (c : listRefElement) do
40:                 if (listExcWithMandatory.contains(c.getName())) then
41:                     error ← "EXCLUSION_REQUIREMENT"
42:                     listErrors.add(error)
43:                 end if
44:             end for
45:         end if
46:     end if
47: else
48:     list = root.getListComponents()
49:     if (list ≠ null)  then
50:         for (c : list) do
51:             checkExclusionMandatoryElement(c)
52:         end for
53:     end if
54: end if
```

6.3.8 Cardinality

The following algorithm ensures that the cardinality of OR nodes is respected.

Algorithm 7 checkCardinality(Component root)

Data : Variability Model
Results : Rule 5 (The number of mandatory children of an OR cannot exceed the OR cardinality) is checked for the variability model

```
1:  List < Component > list ← root.getListComponents();
2:  if (list ≠ null) then
3:      if (root instanceof OR) then
4:          nbrComponent ← 0
5:          nbrMandatory ← 0
6:          nbrOptional ← 0
7:          nbrElement ← list.size()
8:          minCardinality ← root.getCardinality([0])
9:          maxCardinality ← root.getCardinality([2])
10:         mandatoryComponents = ""
11:         for (comp0 : list) do
12:             optional ← false
13:             if (comp0 instanceof Variant) then
14:                 optional ← comp0.isOptional()
15:             end if
16:             if (comp0 instanceof VariationPoint) then
17:                 optional ← comp0.isOptional();
18:             end if
19:             if (≠ optional) then
20:                 nbrMandatory + +
21:                 mandatoryComponents ← mandatoryComponents + comp0 + ";"
22:             else
23:                 nbrOptional + +
24:             end if
25:             nbrComponent + +
26:             checkCardinality(comp0)
27:         end for
```

```
28:         if (nbrMandatory > maxCardinality) then
29:             error ← CARDINALITY_MANDATORY
30:             listErrors.add(error)
31:         end if
32:         if ((nbrMandatory == maxCardinality)&&(nbrOptional > 0)) then
33:             error ← CARDINALITY_MANDATORY_EQUAL
34:             listErrors.add(error)
35:         end if
36:         if (nbrComponent < minCardinality) then
37:             error = CARDINALITY_MIN
38:             listErrors.add(error)
39:         end if
40:     else
41:         for (comp : list) do
42:             checkCardinality(comp)
43:         end for
44:     end if
45: end if
```

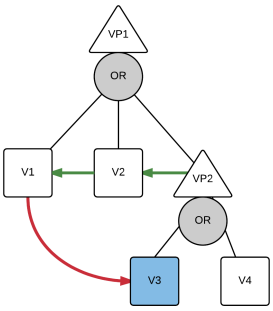

Figure 6.14: Circuit + Mandatory child

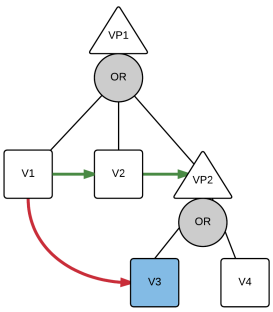

Figure 6.15: Transitivity + Mandatory

6.4 Mapping Features to Logical Schema

According to our methodology, we now have to detail how we map the database schema elements to the variability model elements.

As far as database elements are concerned, we choose to work at the logical schema level. Indeed, a conceptual schema, while it is more explicit of the reality of a domain, needs to be translated in order to get the concrete table and columns of the database. Meanwhile, a physical schema is not the best structure to work with, as it does not show the logical structure of its elements. We thus favored the logical level: table and columns.

For the variability model elements, we need a generic element to express general characteristics, and that can be later broken down into more precise types. According to the Simple Variability Language specification, we define core elements, variation points and variants.

Now that the two structures are defined, we can explain our mapping structure. The mapping structure is based on two lists.

First, the MainList, which contains Tables, which themselves contain Columns. The MainList contains every tables (and for each of them every column) of the database.

Secondly, the ComponentList contains Components, which may be core elements, variation points elements or variant elements. A Component includes zero to many Tables, while a Table can be listed in one to many Components. Similarly, a Component includes zero to many Columns, while a Column can be listed only in one Components. Furthermore, if a Component includes a Column, it also includes the Table this Column belongs to. Figure 6.16 shows an overview of our mapping data structure.

Algorithm 8 displays the algorithm used to check if a Component including a foreign key has the referenced table and index in its ancestors.

Example

We illustrate here the step of mapping database schema elements to the variability model with the schema and the variability model previously presented in section 6.2. By default, all elements are loaded in the Core. Then, tables are moved to the right component. In this case, the schema was decomposed in several groups of tables and then added to the rights variants as shown in figure 6.17.

Algorithm 8 Foreign key verification

Data : an Attribute part of a Component which includes a foreign key Result : true if the referenced table and attributes are in the Component's ancestors

```
1: if attribute.getReferencedIndexTable() ≠ null then
2:     index ← attribute.getReferencedIndexTable()
3:     if (not Component.getMapTables().contains(index) then
4:         if not isCheckAscendant(getParent(),index) then
5:             Erreur
6:         end if
7:     end if
8: end if

9: procedure IsCheckAscendant
10:    if (currentNode.getUserObject.getCommen ≠ null) then
11:        if Common.contains(index) then
12:            return true
13:        else if (currentNode.getParent() ≠ null) then
14:            if (currentNode.getParent().getParent() ≠ null) then
15:                return isCheckAscendant(currentNode.getParent().getParent())
16:            else
17:                return false
18:            end if
19:        else
20:            return false
21:        end if
22:    else
23:        return isCheckAscendant(currentNode.getParent().getParent())
24:    end if
25: end procedure
```

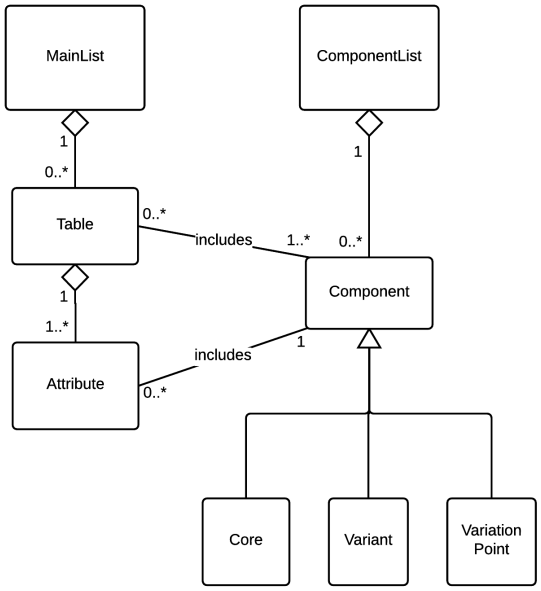

Figure 6.16: Mapping Data Structure

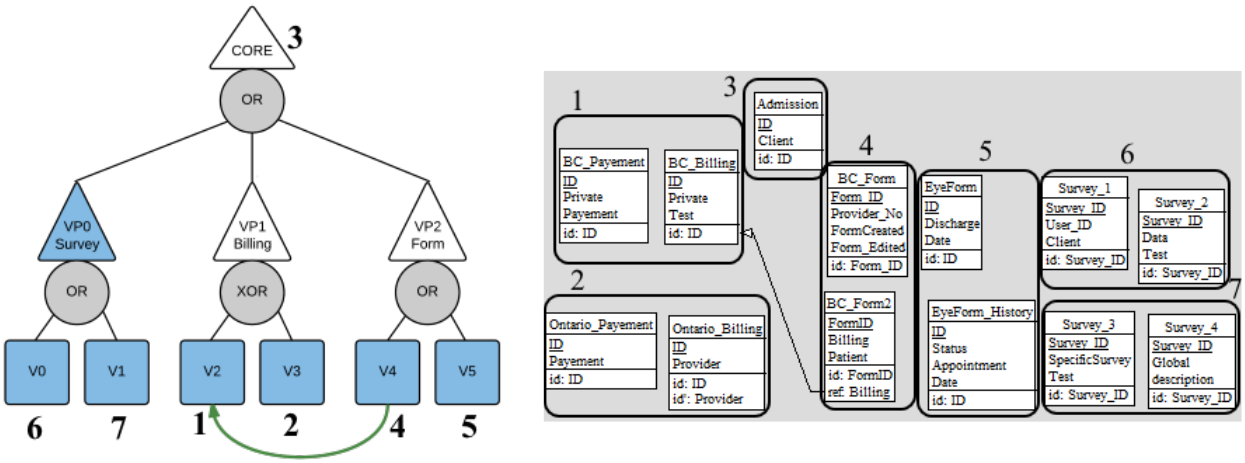

Figure 6.17: Illustration of the mapping

6.5 Database Schema Filtering

In this section we detail the process of adapting a database schema.

6.5.1 Resolution model

The previous step has created a variability model mapped with a base model which is the database schema. The next step consists in the creation of a resolution model, in other words, the selection of a configuration in the model.

6.5.2 Configuration checking

One of the prerequisites for a correct database schema adaptation is that the configuration (a specific fixed version issued from a general variability model, already known as a resolution model) is correct. There is a number of rules that are checked to make sure that the configuration created is correct and thus the filtered database schema will not be incorrect.

1. **Requires/excludes** :
   - the selection of an element implies the selection of all the elements it requires
   - The selection of an element forbids the selection of any element it excludes
2. **Cardinality** :
   - Number of mandatory OR children cannot exceed cardinality
   - Number of selected OR children cannot exceed cardinality
3. **Foreign keys:**
   - If an element includes a foreign key, the referenced table and the referenced identifier have to be in this element's parents or its ancestors' common.

6.5.3 Database schema adaptation

Algorithm 9 explains the process of filtering the database schema.

Example

We illustrate here the step of filtering a database schema based on a selected configuration with the model shown at figure 6.17. A potential configuration for this model is presented at figure 6.18, where the selected elements are highlighted with bold borders. The result of this selection is a filtered database schema as shown on figure 6.19.

Algorithm 9 Database schema adaptation

Data : a database schema and a resolution model for this schema
Results : an adapted database schema

```
1:  entity ← firstEntity
2:  while entity not null do
3:      if entity not in the resolution model then
4:          delete this entity
5:      else
6:          attribute ← firstAttribute
7:          while attribute not null do
8:              if attribute not in the resolution model then
9:                  delete this attribute
10:             end if
11:         end while
12:         attribute ← nextAttribute
13:     end if
14:     entity ← nextEntity
15: end while
```

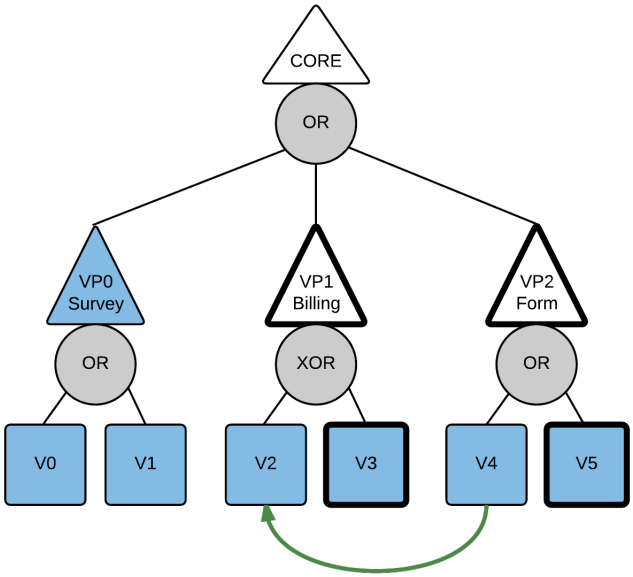

Figure 6.18: Selected configuration

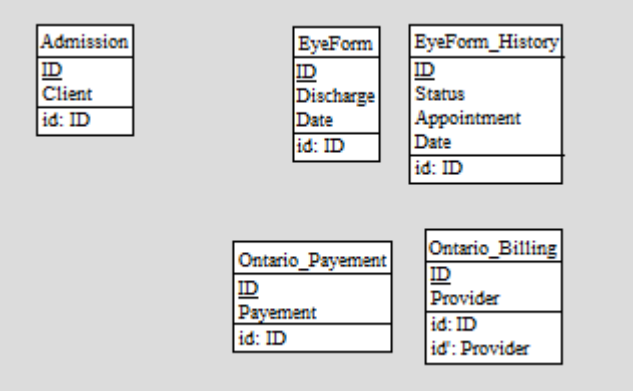

Figure 6.19: Schema obtained with the selected configuration

Chapter 7

Implementation

In the chapter 6, we discussed the design of our solution including data structures, algorithms, patterns, etc. In this chapter, we present an implementation of the solution based on what was described in the design part. Using Java, the Java API JIDBM provided in the DBMain installation to manipulate the .lun files and Swing to design the GUI, we implemented a tool that would handle the three phases of our approach. A screenshot of our tool is available at Figure 7.1.This tool is called SVL Tool and is used as a plugin for DB-MAIN. The tool window is divided into four areas.

- The explorer displays the variability models in the current repository;
- The second pane shows the Variability Model;
- The third pane contains the database schema elements being selected;
- The bottom pane shows the properties for a selected element in the variability model, such as the type of the element, the name, the cardinality if it is a connector, require and exclude constraints if it is a variant or a variation point. It also displays the problems encountered after a check of the model.

The whole workflow of the tool (that can be seen in Figure 6.1) consists of four steps:

1. Creation of the variability model;
2. Mapping of the feature elements to the database schema elements;
3. Instantiation of the generic variability model into a precise configuration;
4. Filtering of the database schema.

Each of these steps is detailed below.

7.1 Variability Modeling

The variability model is created within SVL Tool, which is launched from DB-MAIN. Before creating a variability model, the user has to open a database schema in the DB-MAIN editor. Once the schema is displayed, the user can launch SVL Tool and create a new model. The schema previously open will be loaded in the core. If the user works

on an existing variability model, there is no need to open a database schema as all the information is saved in an XML file. SVL Tool displays the variability models in a "folder-like" view. Figure 7.1 shows a typical screen of SVL Tool.

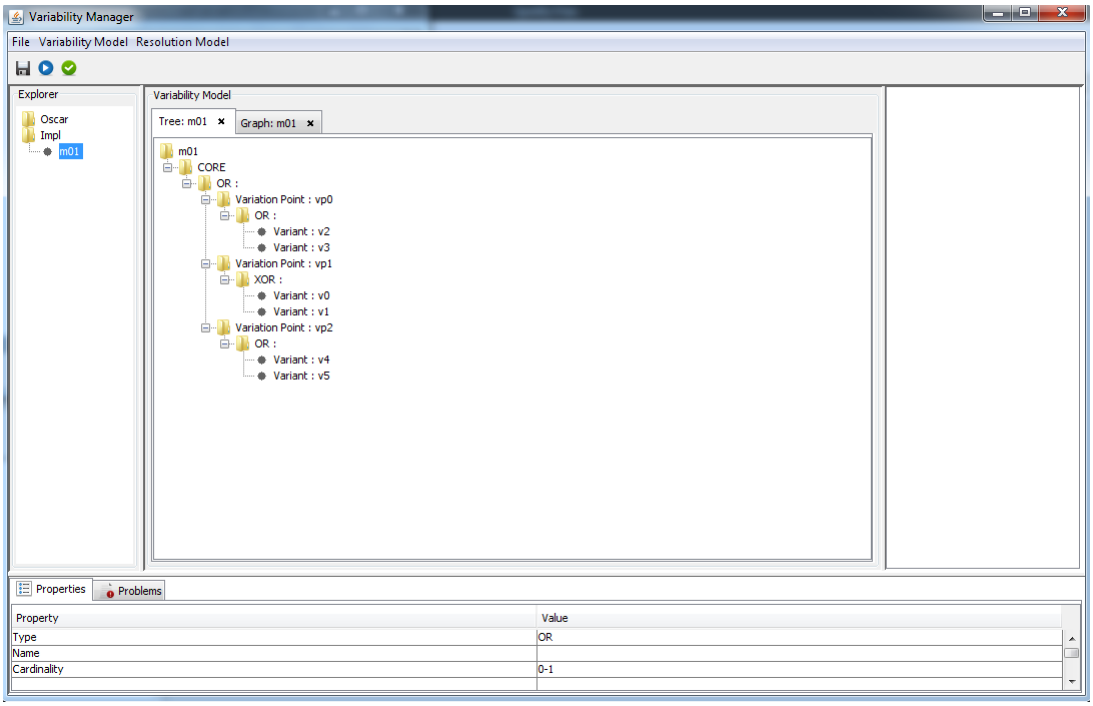

Figure 7.1: Variability model in SVL Tool with the folder View

SVL Tool also provides a secondary view, which displays the variability model as a tree graph. This view is more similar to feature models used in FODA, and offers the same features as the first view. Figure 7.2 illustrates the secondary view. The advantage of this second view is to allow the user to more easily understand the decomposition of the variability model. However, the folder view is more effective when it comes to focusing on a particular branch of the tree because the folders can be expanded or closed to allow readability. Another difference between the two views is that in "folder view", the database schema elements added in a variant are directly visible as another folder. In the "graph view", the choice was made to not create children nodes for the variants with database schema elements to maintain a clear representation of the variability model and not overload it with information from the mapping. Nevertheless, the user has the possibility to see the elements related to a variant in the right panel. The user can then work with both views when building the model.

As detailed earlier, the core includes all the database schema elements that are not included anywhere else. When the user creates a variability model, the core is automat-

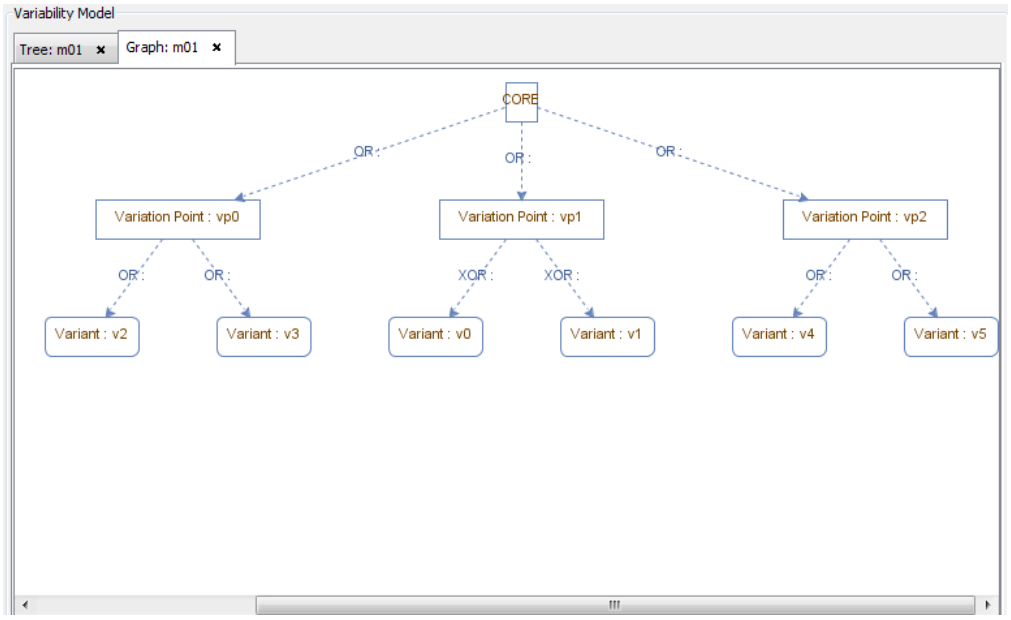

Figure 7.2: SVL Tool Graph View

ically generated and includes all the tables (and their columns) of the current schema displayed in DB-Main. The core is considered as the root of the model. Once the core is loaded, the user can create the whole hierarchy of his model by adding connectors, variation points and variants. When adding theses last two elements, a unique name is automatically generated and attributed . This way, there is no confusion between the different nodes in the model. The tool only allows actions respecting the SVL rules, as they are detailed in the previous chapter. Each time the user selects an element in the model, whether it is in the "folder view" or in the "graph view", the property panel is refreshed with the related information. Then, the user can edit the name while keeping it unique or an error will appear, and add new constraints as explained in the following section 7.2.1.

The whole repository with all its projects and variability models can be saved and loaded at any time. SVL Tool produces and loads XML files.

7.2 Constraints

7.2.1 Constraints addition

Although a variability model is built following the SVL rules, it may be necessary to add new constraints. Some are intrinsic at the chosen component. For instance, a XOR connector implies a mutual exclusion of its children nodes. However, some other

constraints have to be added manually by the user.

A user can add requirements or exclusion constraints by selecting an element. A window then displays the elements that can be added, according to the type of constraints previously chosen.This window is presented at figure 7.3 for the requires constraints. The same window is available for the exclude constraints. A basic safety mechanism ensures that:

- the current element cannot require an element if this element excludes the current element or if the current element excludes this element
- the current element cannot exclude an element if this element requires the current element or if the current element requires this element

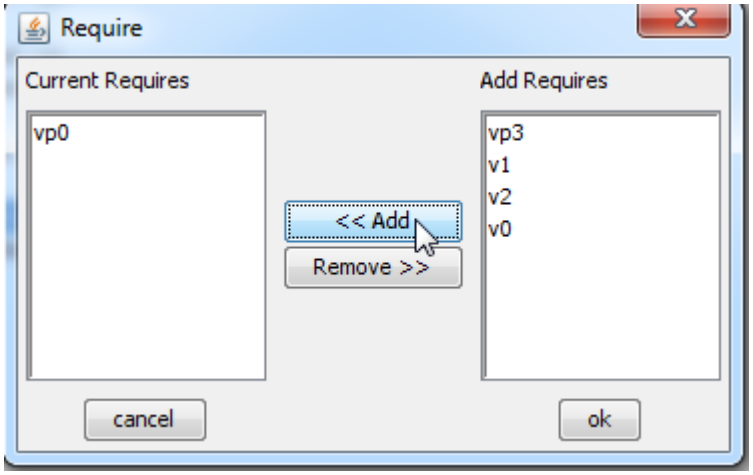

Figure 7.3: Modification of requires constraints

In the property panel, after clicking on a variant or a variation point, it is possible to edit a field called "optional" by setting it to "true" or "false". If it is "true", the variant or variation point is optional. It means that if the parent node is selected during the product generation phase explained in 7.4, it is not mandatory to select the current node. If the user set the field to "false", the current node is mandatory. Thus, if the parent node is selected during the product generation phase, the current node has to be selected.

The last constraint that can be modified is the cardinality of a connector. By default, the cardinality is set to 0-1. The property panel is shown at figure 7.4.

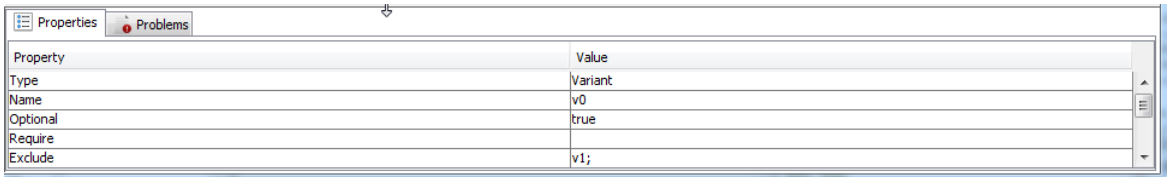

Figure 7.4: Property panel

7.2.2 Constraints checking

A variability model is created according to the building rules of SVL and constraints are added to this model. However, the use of constraints may lead to inconsistencies in the model. Thus, a constraints checking mechanism was developed to avoid this kind of problem. The constraints detailed in the previous chapter can be checked by the user to ensure the consistency of his model.

By clicking on the blue button, the user start a checking operation that will result in a list of problems found in the current variability model. This list is displayed in the bottom panel called "problems". This panel has a table with a column "description" that describe the kind of problem, a column "source" that gives the sources components of the problem and a column "type" that says if it is an error or a warning.

The user has to modify the model to resolve the errors shown in the panel. As long as the model contains errors, it is inoperable and it is not allowed to go to the next phase.

We illustrate the constraints checking mechanism with an example. We decide to add some require and exclude constraints on the model presented in the figures 7.1 and 7.2. The constraints are the following: Variant 2 requires Variant 1 and Variant 4 requires Variant 2 but excludes Variant 1. By transitivity, Variant 4 should require Variant 1 but it is excluded. This is a contradiction in the constraints and it is detected with the algorithm 4 presented in the Design chapter. The result is shown at figure 7.5.

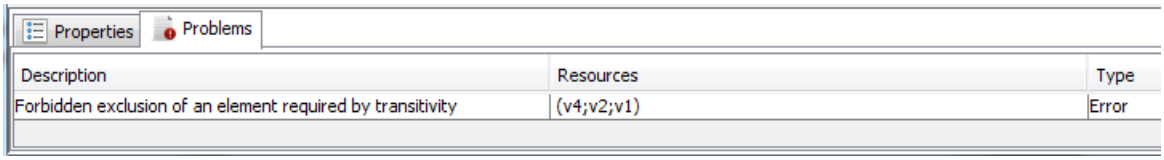

Figure 7.5: Result of constraint checking in the Problem panel

7.3 Mapping Features to Logical Schema

For the mapping of the variability model to the database schema elements, the user has to select a variant or a variation point in the variability model and then search for the database elements he wants to map it to.

Finally, the database element and the variability model element are mapped to each other. The following figures illustrate the process of adding database elements to the variability model.

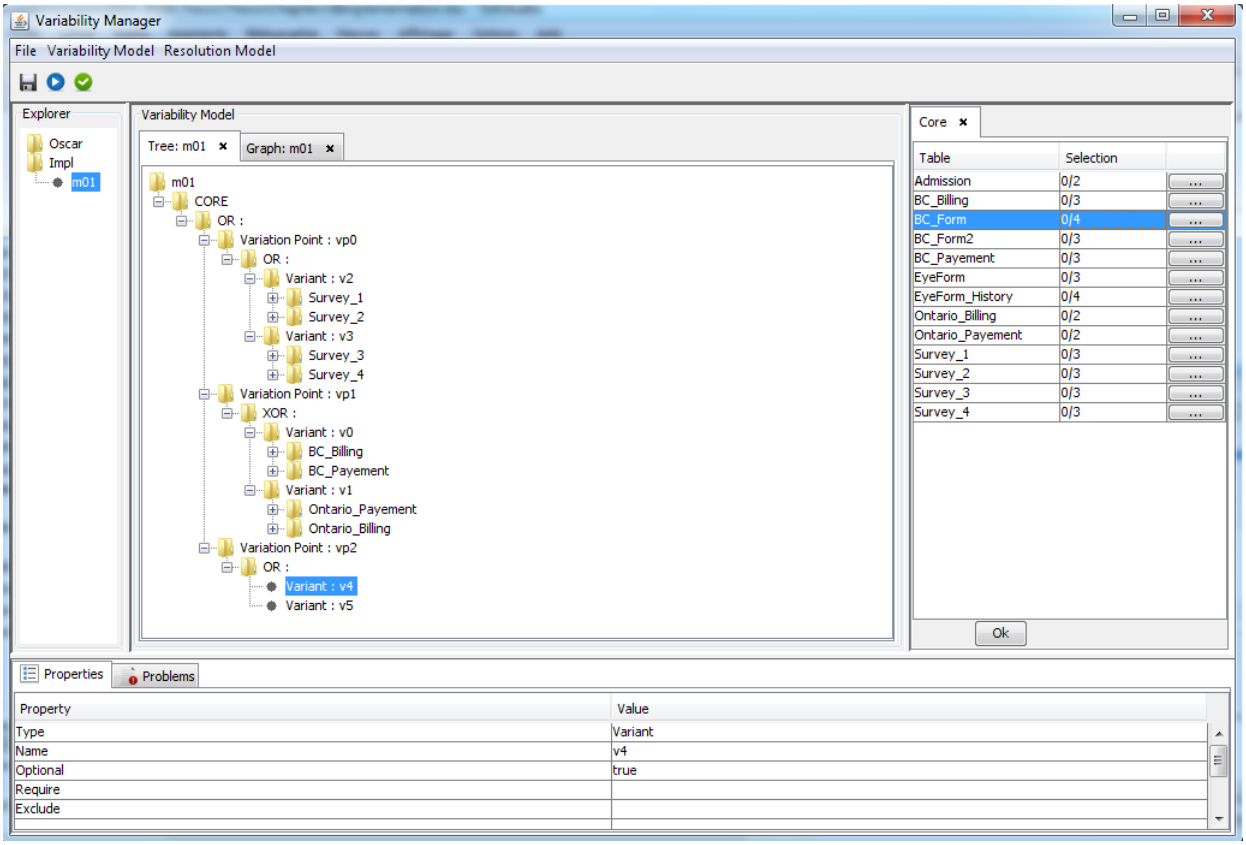

Figure 7.6: Mapping process in SVL Tool

7.3.1 Foreign keys indications

If the user includes a foreign key, the referenced table and column are displayed and a note informs the user that the variant or variation point that will include these elements will be required. This approach makes sense as our user is expected to be a software engineering familiar with database concepts. By adding this constraint, we ensure that the

derivation of the new database schema will be consistent. The foreign keys consistency is checked as a require constraint. This mechanism is presented at figure 7.7

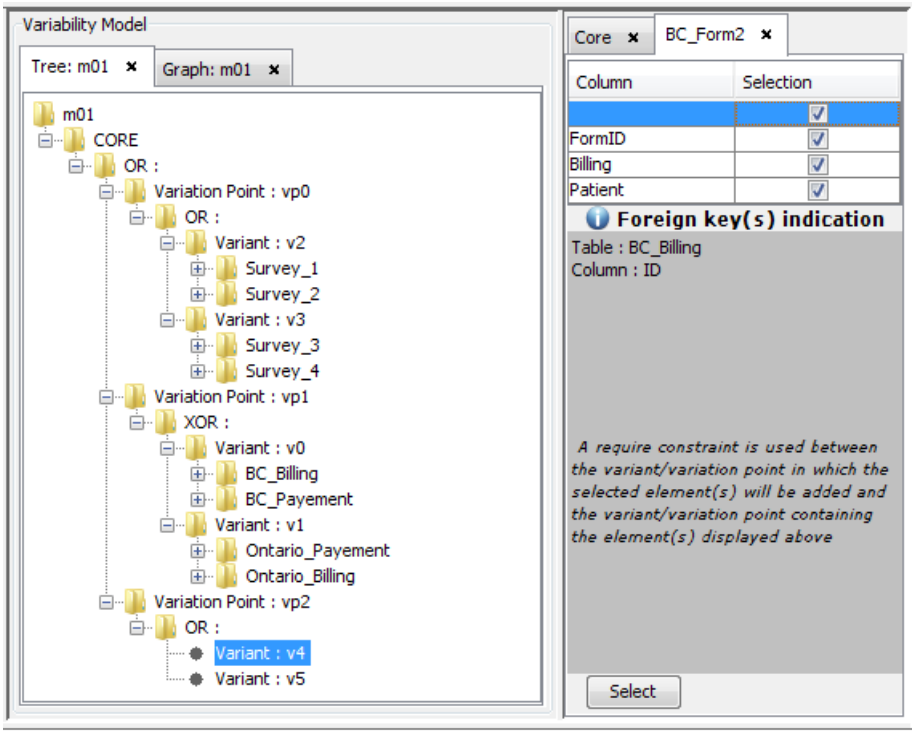

Figure 7.7: Indication of a foreign key in SVL Tool

7.4 Database schema filtering

The precise instantiation starts from the "generate configuration" button of the generic feature model. This creates a model identical to the variability model, except that, in folder view, it contains checkboxes where a choice is required.In the graph view, a right click on the nodes allows to select them in the configuration. The tool implements constraints checking preventing the user to select a configuration including errors. Figure 7.8 presents a screenshot of our tool in a folder view and figure 7.9 presents the selection of a configuration with the graph view.

Finally, the process of filtering the database schema starts from the configuration and produces a filtered database schema.

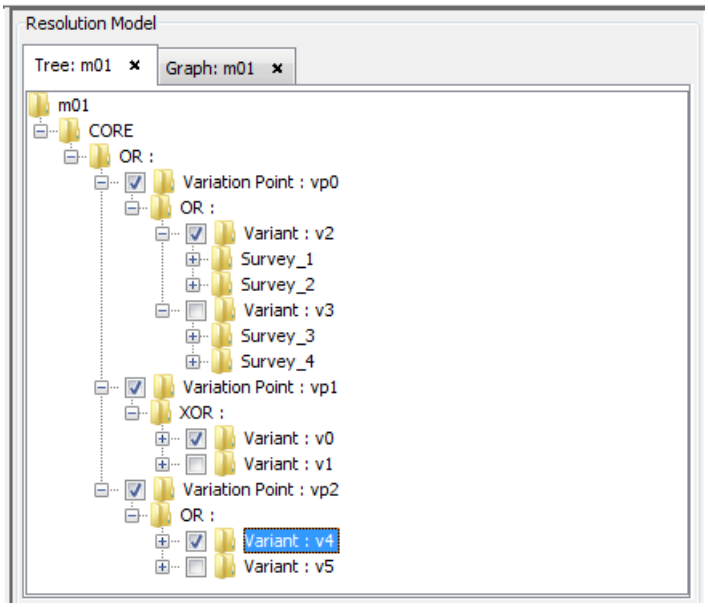

Figure 7.8: Selection of a configuration in the folder view

Now that we have detailed the implementation of our tool, it is time to test it on the OSCAR system. Chapter 8 is thus dedicated to a case study.

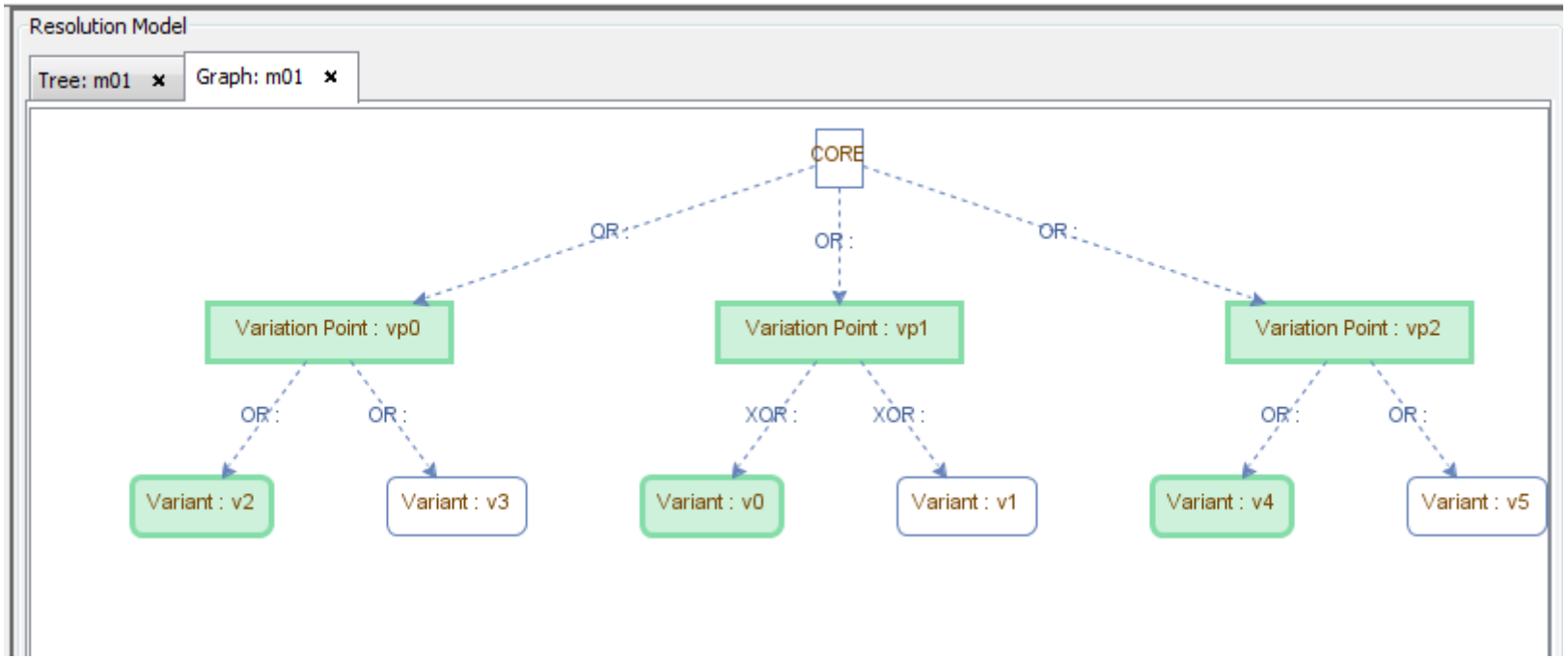

Figure 7.9: Selection of a configuration in the graph view

Chapter 8

Case study : Variability Management in OSCAR

In this chapter, we present the application of our tool in a concrete case. This case was previously introduced in chapter 4 and is discussed more precisely here.

8.1 Preparation

In this section we present the logical database schema of OSCAR that then use in the tool, then a simplified variability model to implement in the tool and finally we define the goal we would like to encounter with SVL Tool and this case study.

8.1.1 OSCAR Database schema

OSCAR has a consistent database and is thus a really interesting case study. We decided to use a logical database schema of OSCAR updated for the last time in 2012 by previous students. This schema is the most complete we have at our disposal and contains additional information such as the expression of implicit foreign keys into explicit ones relevant for the use we make of this schema. In section 7.3 we explain the reason to use a logical schema instead of a physical one.

The schema is presented in figure 8.1 and contains 455 tables.

8.1.2 Goal

Starting from the schema shown at the figure 8.1, we would like to create a variability model in SVL Tool, then select a specific configuration and filter consequently the schema with the selected configuration.

The variability model proposed is a simplified model. Indeed, we are not OSCAR experts and the step of variability extraction was not realised in a systematic way. However, we acquired sufficient knowledge of the system to be able to realized a variability model for which a selected configuration could have a significant impact on the database schema. This model is presented in the figure 8.2 and is the basis for the creation of the full model in our tool with which we integrate constraints that are not represented such as cardinality, the requirements and exclusions and we make the mapping step.

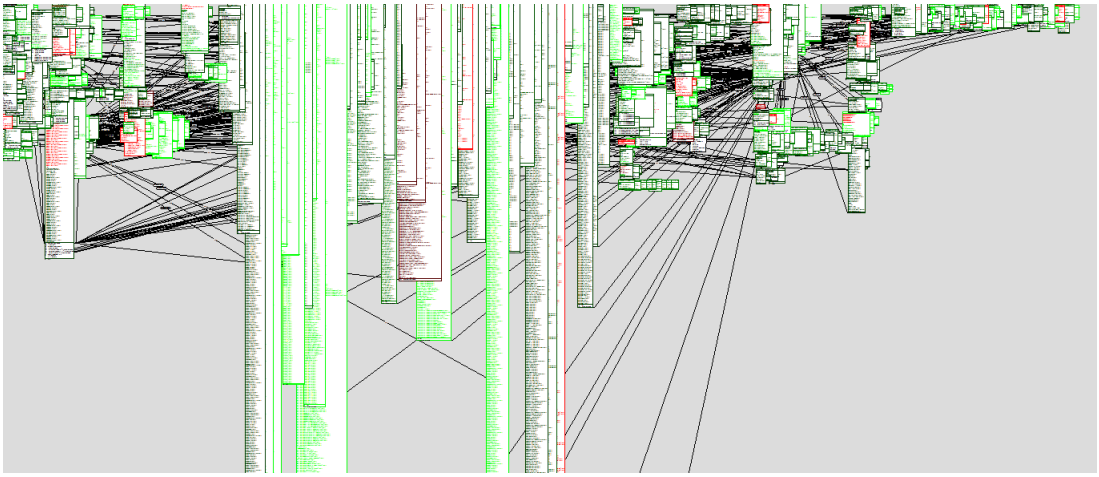

Figure 8.1: Complete OSCAR schema

To have a better understanding of the operations that are executed in SVL Tool, we briefly explain the model. The main variability elements we find in this model is between two Canadian provinces, British Columbia(BC) and Ontario. Indeed, in the database of OSCAR, several tables are only dedicated to BC and other only for Ontario, but that is not all, for example, the variant Surveys is not related to a province but is part of this variability model because it could be optional to have the elements of this variant in an implementation of OSCAR. Thus, the model contains several variation points and variants.

Firstly, the variation point Billing has two variants, BC Billing and Ontario Billing separate by a XOR. It means that we can only select one of the two elements. Indeed, if the goal is to implement the OSCAR system whether it is in BC or Ontario, there is no need to have the two types of billing system. Then we have the Forms variation point with three variants all optional. The last variation point is Labs and contains two variants with the same distinction as for Billing, BC or Ontario labs. Surveys is an optional variant. In green, we have the elements needed for a BC implementation of OSCAR and, in blue, an Ontario implementation. This visual representation is expressed with require and exclude constraints in SVL Tool.

Finally, we have to choose a configuration to filter the database schema. For this case study, we opt for an OSCAR implementation in Ontario with all forms except ones from BC and without the surveys.

8.2 Results

We present here the results of our work on OSCAR using SVL Tool.

8.2.1 Variability Model

After creating a new project and a new variability model in SVL Tool, which leads to load elements of the current database schema displayed in DB-Main in the CORE of the model, we developed the whole variability model base on the simplified one presented

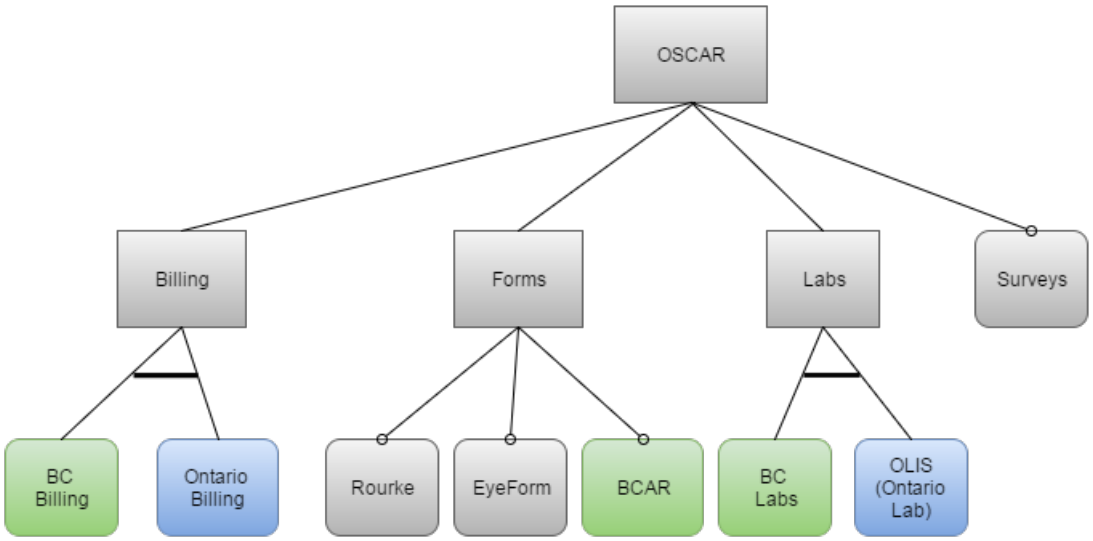

Figure 8.2: Simplified OSCAR variability model

at figure 8.2. The result is shown at figure 8.3 and figure 8.4 with respectively the tree view and the graph view.

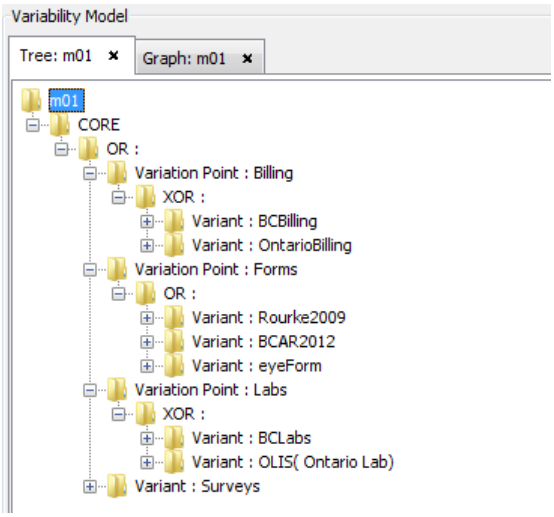

Figure 8.3: OSCAR variability model in SVL Tool

After this, a set of constraints was added to complete the model and are presented here:

- Billing, Forms and Labs were set to mandatory. The other elements were set to optional by default.
- To maintain consistency in a future configuration selection in this model, require and exclude constraints were added.
- Finally, the cardinality of each connector is updated.

Once the variability model totally complete, a check operation was made to ensure its correctness.

The last step for the model is to make the mapping with the database schema elements. We detail here a list of elements mapped with variants :

- OntarioBilling : billing_on_3rdpartyaddress; billing_on_payment;
- BCBilling : billing_private_transactions;
- Rourke2009 : formRourke; formRourke2009;
- BCAR2012 : formBCAR; formBCBirthSumMo; formBCClientChartChecklist‘; formbchp; formBCINR‘; formBCNewBorn;
- eyeForm : eyeform; eyeform_followup ; eyeform_macro; eyeform_macro_billing; eyeform_macro_def; eyeformconsultationreport; eyeformfollowup; eyeformocularprocedure; eyeformprocedurebook; eyeformspecshistory; eyeformtestbook;
- BCLabs : labrequestreportlink;
- OLIS(Ontario Lab) : providerpreference; providerlabrouting;
- Surveys : surveydata; survey_test_instance; survey_test_data; survey;

As previously said, we are not OSCAR expert, thus, this model does not contain all elements for each variant and cannot considered as totally complete. However, this model could represent a simplified version of the reality and is a good base to put to the proof SVL Tool.

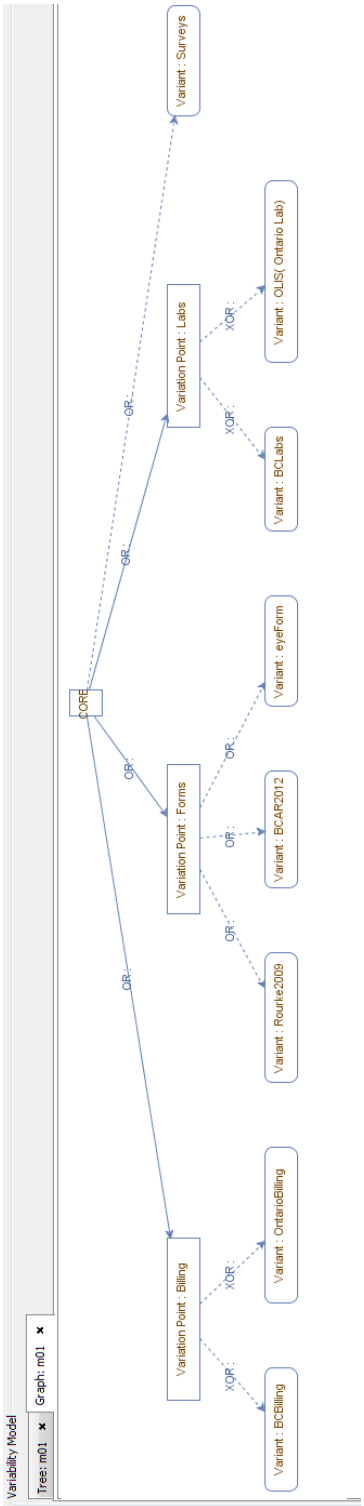

Figure 8.4: OSCAR variability model in SVL Tool

8.2.2 Configuration Selection

Once the variability model has been finished and the mapping was done, we selected the configuration desired. The result is shown at figure 8.5.

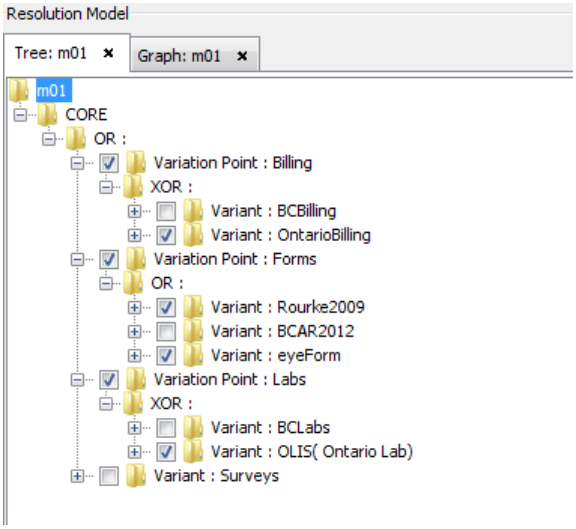

Figure 8.5: Configuration selection

This configuration respects all the constraints, then it is possible to create a resolution file with the given configuration.

8.2.3 Database schema filtering

Once the configuration selected and the resolution file created, we filtered the database schema. The result was a new schema without the variants not present in the resolution model. All the tables from BCBilling,BCAR2012, BCLabs and Surveys were removed from the initial schema. As the schema has a consequent number of tables, it is difficult to perceive all these modifications. Therefore, we have highlighted two groups of tables from two distinct variants, eyeForm, which was selected and BCAR2012, which was not. The figure 8.6 presents the initial OSCAR schema with the two highlighted groups of tables. The figure 8.7 presents the new OSCAR schema after filtering with tables from eyeform variant present.

8.2.4 Conclusion

We have succeeded with SVL Tool to reduce the complexity of managing the variability of a large system like OSCAR. Indeed, we have reduced the number of steps required to get to the final result which is a consistent database schema that matches the needs of the user while simplifying the overall approach.However, everything is not perfect and some elements have to be improved as discussed in chapter 9.

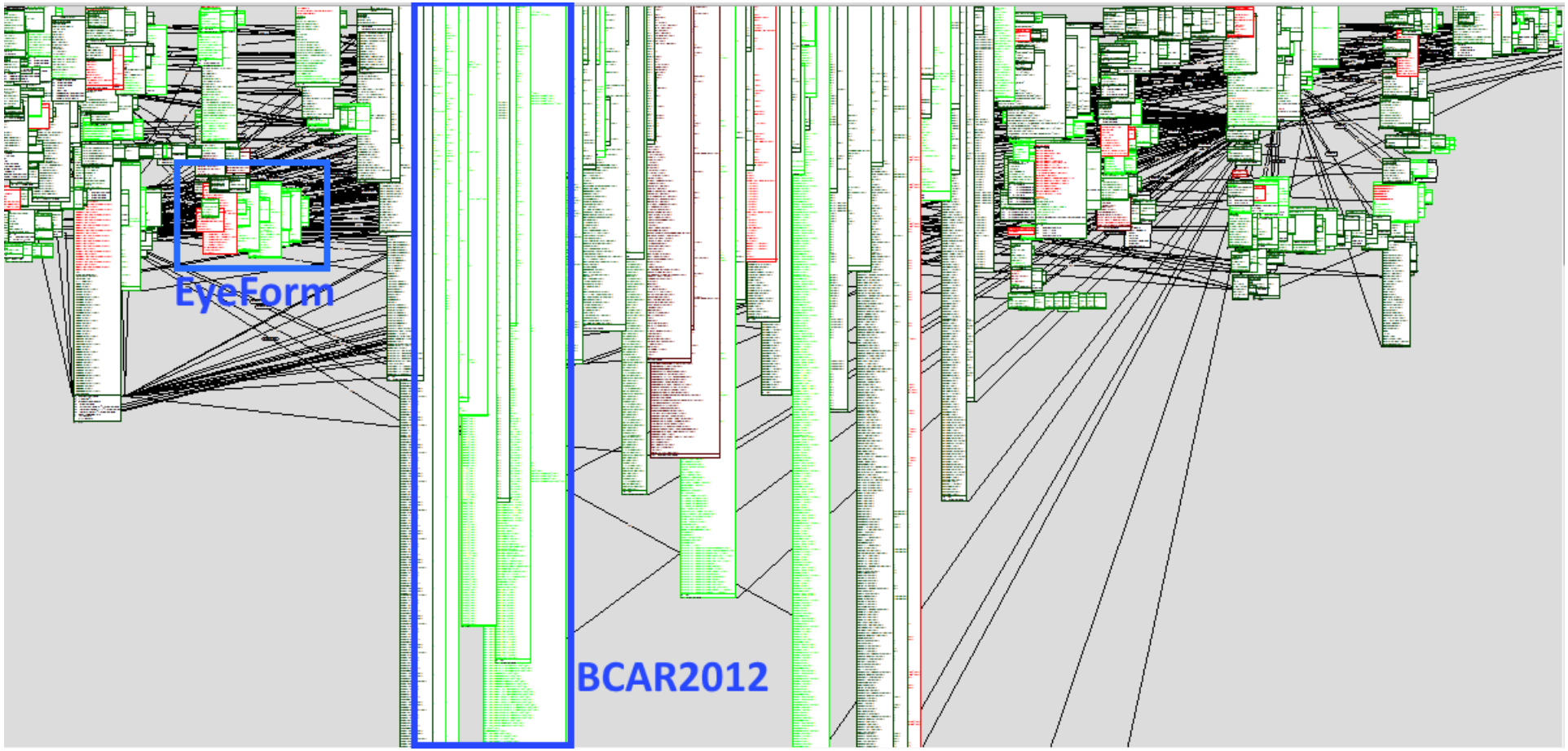

Figure 8.6: Initial OSCAR schema

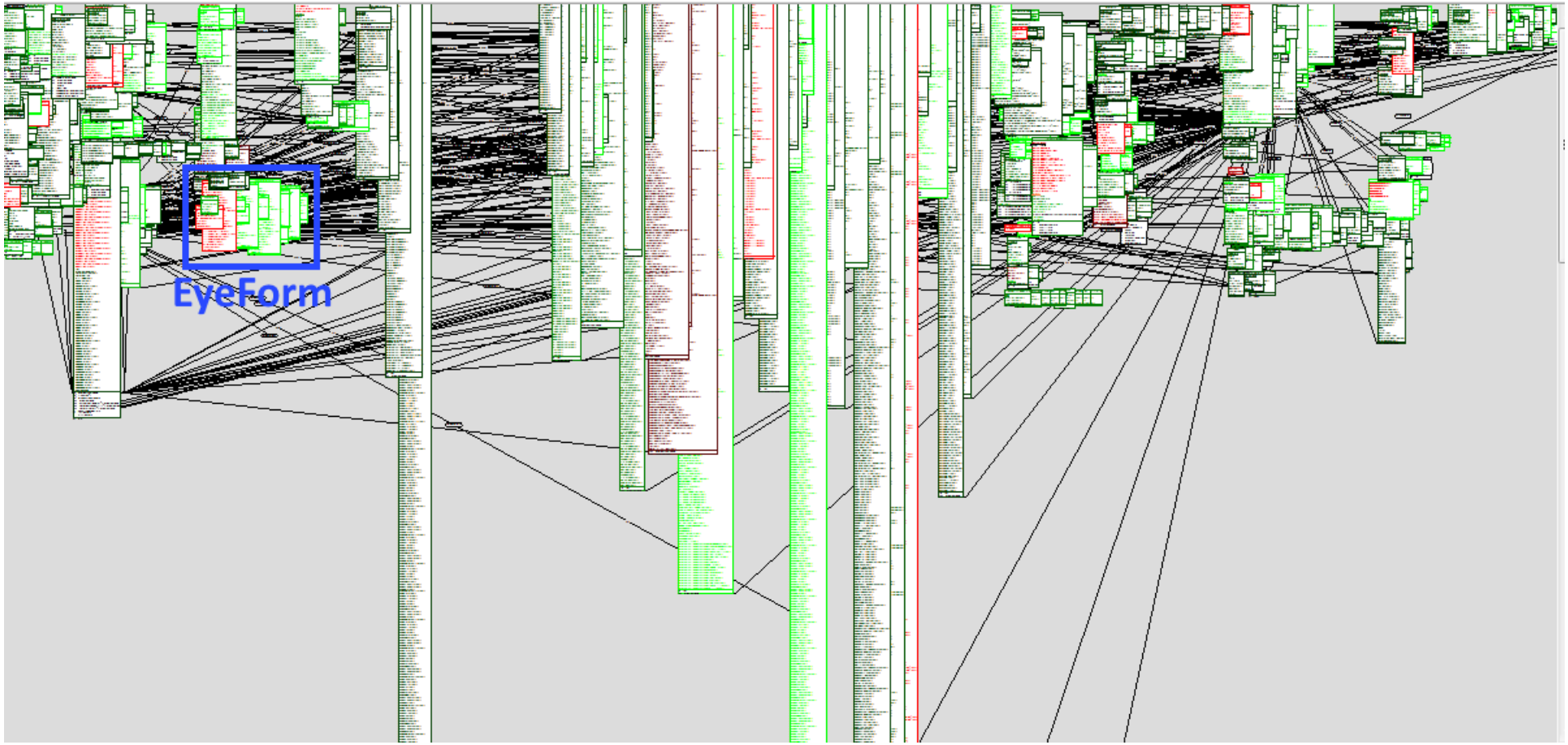

Figure 8.7: OSCAR schema after filtering

Chapter 9

Additional Discussion

In this chapter,we provide additional discussion material concerning our work. The first section contains some of our observations on our approach. It is followed by an analysis focused on the application of our approach. Finally, we discuss the limitations we encountered in the technical implementation of our work.

9.1 Methodology

We detail here some aspects that give an additional perspective to our high-level methodology.

9.1.1 Additive and Subtractive Approaches

During the methodology development, we chose to use a mix between the additive and subtractive approaches for the database schema adaptation. The main advantage of the additive approach was to load all the database schema elements into the core, so there was no need to develop the whole variability model. For the subtractive approach, it was the possibility to work only on a subset of tables, in an incremental approach, for instance.

Our variant of the subtractive approach offers the advantage of loading the whole database schema to have the global overview of the system. There was also an influence from the technical aspects of our work, further information being provided in the following sections.

9.1.2 Users Satisfaction

In our methodology, we focused on the adaptation of database schemas following a variability implementation process, considering the adapted schema as our main type of results. However, as this process is carried out by users, another interesting input could be the global user satisfaction using our methodology and tool.

We can also note that as there are two types of users for our tool (the software engineers and the domain users), the result may vary depending on the population to focus on the satisfaction study.

9.2 Methodology Application to OSCAR

We precise here some limitations that are present in the application of our work on the OSCAR system, and detail the reasons for their existence.

9.2.1 OSCAR Users

One lacking aspect of our work is that we did not currently have the opportunity to test the tool with OSCAR experts and/or users. Despite OSCAR being widely used in Canada, we were not able to meet a representative panel of them, mostly because of time constraints.

9.2.2 OSCAR Complete Variability Model

As we have stated before, OSCAR is a hidden software product line, with a great level of variability. From this, it was not possible when we were there to have a complete variability model of the whole system. We based our work on a re-engineered variability model, which is thus enough to see our work as a proof of concept, but we could not test our methodology and tool on the complete system.

9.3 Implementation

After having discussed the high level and the application of the methodology, we list here our main comments on the technologies we used for the implementation of our work and we also discussed some features that would have been interesting to implement in the tool.

9.3.1 Swing and DB-MAIN

Swing is a GUI widget toolkit for Java. It is an API for providing a graphical user interface (GUI) for Java programs. The choice was made to use this API to create the GUI for SVL Tool. Firstly because it is easy and fast to create simple GUI. Secondly, because we wanted to develop a plugin that could be launched from DB-Main. The reason for this was to have a direct access to the current database schema displayed in DB-Main, then load it in our plugin SVL Tool and perform operations that would have been directly visible on the schema. However, these functionalities required heavy computations and after a few tests, we came to the conclusion that it was not practical, too resource intensive and this kind of functionality would not have been usable with schema such as the one from OSCAR.

Since then, the necessity of launching SVL Tool from DB-Main to have access to the currently displayed schema disappears. Therefore, is Swing a legitimate candidate for the GUI? As mentioned earlier, Swing is great to create a simple application but when it comes to more sophisticated interfaces, it can be really burdensome to obtain the result wanted. Thus, the development of a web application could have been a good solution. Indeed, there is a lot of languages and frameworks dedicated to the development of web interfaces.

9.3.2 JAXB and XML files

As we wanted our exported files to be compatible with other standards, we used the XML language to export them. Aside from the size of the files generated with large database schemas (an average of MB), the major drawback of this choice is that JAXB, the main API used to generate XML, does not work well with the DB-MAIN API. We fortunately found a way to circumvent this limitation.

9.3.3 Additional Feature

By using SVL Tool, we discovered that it would be useful to have another feature to facilitate the process. It would have been interesting to be able to cluster some elements from the base model based on their name to make the mapping with the variability model easier. Indeed, if we look at OSCAR, tables specific for British Columbia contain the letters "BC" in their name. Thus, having a feature to make a cluster with all the tables containing "BC" would make the mapping step way easier.

Chapter 10

Conclusion

At the end of this thesis, we can finally address our research questions. We conclude that representing and managing variability in database applications is possible. We detailed in this thesis a methodology and a tool to achieve this kind of task.

In the context of variability management in database applications, the first step is to model the variability. This step aims at representing correctly the potential different versions of a software product. One of the prerequisites of this step is a deep understanding of how the software works, as it is impossible to create a relevant model without it. In our work, we defined the Simple Variability Language with its precise structures and rules, ensuring the consistency of the created model.

The second step of the process is to map this variability model to the database elements. In this thesis, we focused on the database schema elements, as the conceptual level seems to be more appropriate for this kind of task. SVL Tool, the plug-in we created provides a user-friendly way to map database schema elements to variability model elements, and helps the software engineer user in these tasks by sending him alerts and hints on the mapping he is working on.

Once the variability model and the database schema are mapped to each other, the user profile of our tool changes, as the software engineer is now replaced by a user of the domain of the initial database application. This user no longer needs to have a good understanding of how the database of the application is accessed, as this knowledge is now in the mapping. All he needs to do is then to choose on the variability model which variability elements (for him, these would better be defined as *features*). Our plug-in then derives a new database schema, starting from the original schema and selecting only the database elements needed to provide all the features the user selected.

The main advantage of this approach lies in its two phases workflow. Indeed, the first phased is accomplished by the software expert, who is the best qualified to perform the task of mapping, when the second one is done by the domain user. This means that for

a single expert creating one mapping, a multitude of users can create a version of the database that suits their needs.

The second strength of our work is the support it provides, especially to the domain user. We spent a lot of our time on the question of the constraints the mapped model should include, as this was the best way to ensure the filtered database schema could be directly used. Furthermore, we also designed an error notification system, preventing the user to create incorrect schemas. Moreover, as we do not have any requirements concerning the domain user, we designed a tool as easy to use as possible.

Of course, we do not pretend to resolve the whole database variability management issue with this thesis. We had, of course, a few limits, the first one being we do not claim to have understood the whole OSCAR system. We are not OSCAR experts, and even after having worked for three months on it, we still do not consider ourselves as such. The second limit is that our approach is by definition incomplete. We focused on a specific problem for a finite amount of time, while handling the system for the whole variability life cycle would exceed that by far. The last one is that our study was focused on the very precise case of OSCAR. While our methodology can still be applied to every relation database system, the specific characteristics of OSCAR and its user base influenced our work.

As information systems get more complex every day, variability management is one of the solutions to reduce their inherent complexity. As any other software artefact, databases do not escape the rules. In this thesis, we provided a way to represent and manage variability in database applications. With both our methodology and our tool, we provided an original contribution as a new way to approach this question.

Chapter 11

Future works & Applications

We do not claim to have studied every aspect of variability in database applications. Our tool provides a first approach to manage variability on the database level, but is only the beginning of a broader approach if you see this in a global perspective.

11.1 Variability Extraction

We conducted our work with the assumption that the user was in possession of the domain knowledge needed to model variability. One possible improvement of SVL Tool, once it is fully functional, would probably be to add some features of variability extraction. Based on heuristics and detection algorithms, such a system would help the software engineer in charge of implementing variability in OSCAR by providing some variability patterns presents in the database.

11.2 User Expectations Lowering

The second major assumption we made during our research was that the end user of our tool would be a software reengineering expert. That implied he would be able to infer some rules that may not be explicit but are obvious for anyone who has some knowledge on variability modeling. It also led us to not protect every aspect of the variability model creation, as an expert would use his common sense and thus not break the model every two actions.

Another improvement would then be to the addition of safety mechanisms in the creation of the variability model. As our aim was to provide a tool for experts in software reengineering, we had some expectations of the knowledge the SVL Tool user would have. If SVL would be more broadly used, it will require more user-friendly warnings and automation.

11.3 Holistic Approach

As our tool works, it still cannot handle all the variability aspects. It focuses on variability modeling and realization, and leaves the other aspects of the domain (testing, evolution) on the side (with the specific case of variability extraction has already been covered).

The next step of this work could be to go even further into the variability implementation and adapt our methodology to develop a tool working directly with the Java code of the applications. This future tool, used with SVL, would then bring a whole integrated solution for managing variability in database applications.

Another possibility for this improvement could be working directly with the SQL code, and so bypassing the DB-MAIN limitations we are currently experiencing.

11.4 All Database Levels Management

Our work currently focuses exclusively on the logical schema, as this is the level of the database design process that seemed the most appropriate to start our work. One major improvement would be to deal with the physical level too, and finally to be able to work directly with the data itself.

11.5 Multi-repositories Capabilities

One of our first motivations for the job was resolving the problems encountered by different developers on different versions of the same software product. Our tool could be used in a way to improve and help this user by linking the different versions to the variability-enabled database schema.

11.6 Methodology Reuse

During our work, we developed not only a tool, but also a methodology to implement variability in database applications. If our tool may lack some generic character, our approach can fully be reused for other database systems.

Bibliography

[1] *ABOUT OSCAR*. http://www.varies.eu/. Accessed: 2015-10-5.

[2] *ABOUT OSCAR*. http://oscarcanada.org/about-oscar/brief-overview/index_html. Accessed: 2015-10-5.

[3] *Advantages and Limtations of CASE Tools*. http://www.petruska.com/xoops1/modules/AMS/print.php?storyid=15. Accessed: 2015-10-5.

[4] Sven Apel et al. *Feature-Oriented Software Product Lines*. Springer, 2013.

[5] *Appswatch*. http://www.nrgglobal.com/application-testing-tools/performance-testing-and-monitoring-appswatch. Accessed: 2015-10-5.

[6] Wesley K G Assunção et al. "Extracting Variability-Safe Feature Models from Source Code Dependencies in System Variants". In: (2015), pp. 1303–1310.

[7] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. "Feature location for software product line migration". In: *Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC '14* (2014), pp. 52–59. DOI: 10.1145/2647908.2655967. URL: http://dl.acm.org/citation.cfm?doid=2647908.2655967.

[8] Clara Ayora et al. "Applying CVL to business process variability management". In: *Proceedings of the VARiability for You Workshop on Variability Modeling Made Useful for Everyone - VARY '12* (2012), pp. 26–31. DOI: 10.1145/2425415.2425421. URL: http://dl.acm.org/citation.cfm?doid=2425415.2425421.

[9] Jongmoon Baik et al. "The effect of case tools on software development effort". In: *Unpublished Dissertation, USC Computer Science Department* (2000).

[10] Joerg Bartholdt et al. "Addressing Data Model Variability and Data Integration within Software Product Lines". In: 2.1 (2009), pp. 84–100.

[11] Thorsten Berger et al. "A survey of variability modeling in industrial practice". In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '13* (2013), p. 1. DOI: 10.1145/2430502.2430513. URL: http://dl.acm.org/citation.cfm?doid=2430502.2430513.

[12] Thorsten Berger et al. "What is a feature?: a qualitative study of features in industrial software product lines". In: (2015), pp. 16–25. DOI: 10.1145/2791060.2791108. URL: http://dl.acm.org/citation.cfm?id=2791060.2791108.

[13] David Broneske et al. “Software Design Approaches for Mastering Variability in Database Systems”. In: (2014).

[14] Lianping Chen and Muhammad Ali Babar. “A systematic review of evaluation of variability management approaches in software product lines”. In: *Information and Software Technology* 53.4 (2011), pp. 344–362. ISSN: 09505849. DOI: 10.1016/j.infsof.2010.12.006. URL: http://dx.doi.org/10.1016/j.infsof.2010.12.006.

[15] Lianping Chen and Muhammad Ali Babar. “A systematic review of evaluation of variability management approaches in software product lines”. In: *Information and Software Technology* 53.4 (2011). Special section: Software Engineering track of the 24th Annual Symposium on Applied ComputingSoftware Engineering track of the 24th Annual Symposium on Applied Computing, pp. 344–362. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2010.12.006. URL: http://www.sciencedirect.com/science/article/pii/S0950584910002223.

[16] E. F. Codd. *The Relational Model for Database Management: Version 2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0-201-14192-2.

[17] Michael Dillon and Jorge Rivera. “A Methodical Approach to Product Line Adoption”. In: (2014), pp. 340–349.

[18] Bogdan Dit et al. “Feature location in source code: A taxonomy and survey”. In: *Journal of software: Evolution and Process* 25.1 (2013), pp. 53–95. ISSN: 20477481. DOI: 10.1002/smr.567.

[19] *Doxygen*. http://www.stack.nl/~dimitri/doxygen/. Accessed: 2015-10-5.

[20] *Drupal*. http://www.webopedia.com/TERM/D/Drupal.html. Accessed: 2015-10-5.

[21] *Eclipse EPF*. https://eclipse.org/epf/. Accessed: 2015-10-5.

[22] Holger Eichelberger and Klaus Schmid. “A systematic analysis of textual variability modeling languages”. In: *Proceedings of the 17th International Software Product Line Conference on - SPLC ’13* (2013), p. 12. ISSN: 1433-2779. DOI: 10.1145/2491627.2491652. URL: http://dl.acm.org/citation.cfm?doid=2491627.2491652.

[23] Holger Eichelberger and Klaus Schmid. “IVML: a DSL for configuration in variability-rich software ecosystems”. In: (2015), pp. 365–369. DOI: 10.1145/2791060.2791116. URL: http://dl.acm.org/citation.cfm?id=2791060.2791116.

[24] Holger Eichelberger et al. “EASy-Producer – Product Line Development for Variant-Rich Ecosystems”. In: (2014), pp. 133–137.

[25] V Englebert et al. “DB-MAIN: un atelier d’ingénierie de bases de données1, 2”. In: (1995).

[26] *ETL*. http://datawarehouse4u.info/ETL-process.html. Accessed: 2015-10-5.

[27] João Bosco Ferreira Filho et al. “Assessing product line derivation operators applied to Java source code: an empirical study”. In: (2015), pp. 36–45. DOI: 10.1145/2791060.2791099. URL: http://dl.acm.org/citation.cfm?id=2791060.2791099.

[28] John Fitzgerald et al. “Complex Systems Design & Management”. In: (2014), pp. 1–19. DOI: 10.1007/978-3-319-02812-5. URL: http://link.springer.com/10.1007/978-3-319-02812-5.

[29] Jaime Font et al. “Automating the variability formalization of a model family by means of common variability language”. In: *Proceedings of the 19th International Conference on Software Product Line - SPLC ’15* (2015), pp. 411–418. DOI: 10.1145/2791060.2793678. URL: http://dl.acm.org/citation.cfm?doid=2791060.2793678.

[30] José a. Galindo et al. “Debian packages repositories as software product line models. Towards automated analysis”. In: *CEUR Workshop Proceedings* 688.September 2015 (2010), pp. 29–34. ISSN: 16130073.

[31] *GanttProject*. http://www.ganttproject.biz. Accessed: 2015-10-5.

[32] Afifa Ghenai et al. “VARIES VARiability In safety-critical Embedded Systems”. In: 2.5 (2012), pp. 33–48.

[33] Jean-Luc Hainaut. *Bases de données-2e éd.-Concepts, utilisation et développement: Concepts, utilisation et développement*. Dunod, 2012.

[34] Robert Hellebrand et al. “Coevolution of Variability Models and Code : An Industrial Case Study”. In: (2014), pp. 274–283.

[35] Christopher Henard et al. “Multi-Objective Test Generation for Software Product Lines”. In: *Splc* (2013), pp. 62–71. DOI: 10.1145/2491627.2491635. URL: http://doi.acm.org/10.1145/2491627.2491635.

[36] Michaela Huhn and Sara Bessling. “Enhancing Product Line Development by Safety Requirements and Verification.” In: *Fhies* (2012), pp. 37–54. DOI: 10.1007/978-3-642-39088-3_3. URL: http://dx.doi.org/10.1007/978-3-642-39088-3%5C_3.

[37] Ibm et al. “Common Variability Language (CVL) OMG Revised Submission”. In: Cvl (2012).

[38] Stan Jarzabek and Riri Huang. “The case for user-centered CASE tools”. In: *Communications of the ACM* 41.8 (1998), pp. 93–99.

[39] *JavaScript*. http://www.webopedia.com/TERM/J/JavaScript.html. Accessed: 2015-10-5.

[40] Chrls F Kemerer. “Now the learning curve affects CASE tool adoption”. In: *Software, IEEE* 9.3 (1992), pp. 23–28.

[41] Niloofar Khedri. “Handling Database Schema Variability in Software Product Lines”. In: (2013). DOI: 10.1109/APSEC.2013.52.

[42] Hyesun Lee et al. “Experience Report on Using a Domain Model-Based Extractive Approach to Software Product Line Asset Development”. In: *Formal Foundations of Reuse and Domain Engineering* 5791 (2009), pp. 137–149. ISSN: 03029743. DOI: 10.1007/978-3-642-04211-9.

[43] Mark E. McMurtrey et al. “Current utilization of CASE technology: lessons from the field”. In: *Industrial Management & Data Systems* 100.1 (2000), pp. 22–30. DOI: 10.1108/02635570010273027. eprint: http://dx.doi.org/10.1108/02635570010273027. URL: http://dx.doi.org/10.1108/02635570010273027.

[44] Jens Meinicke et al. “An overview on analysis tools for software product lines”. In: *Proceedings of the 18th International Software Product Line Conference on Companion Volume for Workshops, Demonstrations and Tools - SPLC ’14* (2014), pp. 94–101. DOI: 10.1145/2647908.2655972. URL: http://dl.acm.org/citation.cfm?id=2647908.2655972.

[45] Andreas Metzger and Klaus Pohl. *Software product line engineering and variability management: achievements and challenges*. Vol. 40. 3. 2014, pp. 70–84. ISBN: 9781450328654. DOI: 10.1145/2593882.2593888. URL: http://dl.acm.org/citation.cfm?doid=2593882.2593888.

[46] *Mockup builder*. http://mockupbuilder.com. Accessed: 2015-10-5.

[47] Marco Mori and Anthony Cleve. “Feature-Based Adaptation of Database Schemas”. In: (2013), pp. 85–105.

[48] *MySQL Definition*. http://www.motive.co.nz/glossary/mysql.php. Accessed: 2015-10-5.

[49] Simon Niechzial. “Visualization and Analysis of Product-Line Evolution in Codeface”. In: (2014).

[50] Dag Nystrom et al. “COMET: A Component-Based Real-Time Database for Automotive Systems”. In: *In Proceedings of the Workshop on Software Engineering for Automotive Systems* (2004), pp. 1–8. DOI: 10.1049/ic:20040333.

[51] Carlos Parra et al. “Extractive SPL Adoption Using Multi-level Variability Modeling”. In: *Proceedings of the 16th International Software Product Line Conference - Volume 2* II (2012), pp. 99–106. DOI: 10.1145/2364412.2364429. URL: http://doi.acm.org/10.1145/2364412.2364429.

[52] Leonardo Passos et al. “Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel”. In: *Proceedings of the 16th International Software Product Line Conference - Volume 1* (2012), pp. 76–85. DOI: 10.1145/2491627.2491628. URL: http://gp.uwaterloo.ca/sites/default/files/coevolution.pdf.

[53] C Piaszczyk. "Model Based Systems Engineering with Department of Defense Architectural Framework". In: *Systems Engineering* 14.3 (2011), pp. 305–326. ISSN: 1098-1241, 1098-1241. DOI: 10.1002/sys. URL: http://ezproxy.lib.ucf.edu/login?url=http://search.proquest.com/docview/926281441?accountid=10003$%5Cbackslash$nhttp://sfx.fcla.edu/ucf?url%5C_ver=Z39.88-2004%5C&rft%5C_val%5C_fmt=info:ofi/fmt:kev:mtx:journal%5C&genre=article%5C&sid=ProQ:ProQ:civilengineering%5C&atitle=Model+Based+Systems.

[54] K. Pohl et al. *Software Product Line Engineering. Foundations, Principles, and Techniques*. Vol. 49. 12. 2005, p. 467. ISBN: 9783540243724. DOI: 10.1007/3-540-28901-1. URL: http://www.springerlink.com/index/10.1007/3-540-28901-1$%5Cbackslash$nhttp://www.uwplatt.edu/csse/courses/prev/csse411-materials/s11/Burmestera%20-%20Software%20Product%20Line%20Engineering.doc$%5Cbackslash$nhttp://books.google.com/books?hl=en%5C&lr=%5C&id=J4GqT4OUsSMC%5C&oi=fnd%5C&pg=P.

[55] Emmanuelle Rouille et al. "Leveraging CVL to manage variability in software process lines". In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC* 1 (2012), pp. 148–157. ISSN: 15301362. DOI: 10.1109/APSEC.2012.82.

[56] *Rouse,2007a*. http://searchsoftwarequality.techtarget.com/definition/quality-assurance. Accessed: 2015-10-5.

[57] *Rouse,2015a*. http://searchmanufacturingerp.techtarget.com/definition/CASE-computer-aided-software-engineering. Accessed: 2015-10-5.

[58] Hani Safadi et al. "Open-source health information technology: A case study of electronic medical records". In: *Health Policy and Technology* 4.1 (2015), pp. 14–28. ISSN: 22118837. DOI: 10.1016/j.hlpt.2014.10.011. URL: http://linkinghub.elsevier.com/retrieve/pii/S2211883714000847.

[59] Universidade Salvador. "Fault Model-Based Variability Testing (Ph.D. Thesis)". In: (2014).

[60] Ana B. Sánchez et al. "Variability testing in the wild: the Drupal case study". In: *Software & Systems Modeling* (2015). ISSN: 1619-1366. DOI: 10.1007/s10270-015-0459-z. URL: http://link.springer.com/10.1007/s10270-015-0459-z.

[61] Hanae Sbai et al. "A pattern based methodology for evolution management in business process reuse". In: *IJCSI International Journal of Computer Science Issues,* 11.1 (2014), pp. 1694–0814. URL: http://ijcsi.org/papers/IJCSI-11-1-1-211-220.pdf.

[62] Ina Schaefer et al. "Delta-oriented programming of software product lines". In: *Software Product Lines: Going Beyond*. Springer, 2010, pp. 77–91.

[63] *servlet*. http://www.webopedia.com/TERM/S/servlet.html. Accessed: 2015-10-5.

[64] Norbert Siegmund et al. “Bridging the gap between variability in client application and database schema”. In: *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)* (2009), pp. 297–306. ISSN: 1617-5468. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.143.2884%5C&rep=rep1%5C&type=pdf.

[65] Andreas Svendsen et al. “Developing a Software Product Line for Train Control : A Case Study of CVL”. In: *14th International Software Product Line Conference* (2010), pp. 106–120. DOI: 10.1007/978-3-642-15579-6_8.

[66] Andreas Svendsen et al. “Specifying a testing oracle for train stations - Going beyond with product line technology”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7167 LNCS (2012), pp. 187–201. ISSN: 03029743. DOI: 10.1007/978-3-642-29645-1_20.

[67] Maurice H. Ter Beek et al. “VMC: A tool for product variability analysis”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7436 LNCS (2012), pp. 450–454. ISSN: 03029743. DOI: 10.1007/978-3-642-32759-9_36.

[68] *TutorialsPoint*. http://www.tutorialspoint.com/software_engineering/case_tools_overview.htm. Accessed: 2015-10-5.

[69] Frank van der Linden et al. *Software product lines in action - the best industrial practice in product line engineering*. Springer, 2007, pp. I–XX, 1–333.

[70] Jens H Weber et al. “Uncovering Variability Models for Software Ecosystems from Multi-Repository Structures”. In: *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*. ACM. 2015, p. 103.

[71] *Webopedia*. http://www.webopedia.com/TERM/D/data_modeling.html. Accessed: 2015-10-5.

[72] *What is Linux? A Webopedia Definition*. http://www.webopedia.com/TERM/L/linux_os.html. Accessed: 2015-10-5.

[73] Lamia Abo Zaid and Olga De Troyer. “Towards modeling data variability in software product lines”. In: *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2011, pp. 453–467.

[74] B Zhang and M Becker. “Code-based variability model extraction for software product line improvement”. In: *ACM International Conference Proceeding Series* 2 (2012), pp. 91–98. DOI: 10.1145/2364412.2364428. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-84867467640%5C&partnerID=40%5C&md5=7316a031adabd2f23cb2dc3a47a856d6.

Appendix A

Algorithms

We detail here the procedures used in the high levels algorithms.

Algorithm 10 buildDescendant

Data : Variability model

Result : New require constraint added to the variability model

```
1: procedure BUILDDESCENDANT
2:     if (ParentNode.childcount() ≥ 0) then
3:         connector = getFirstChild()
4:         while connector ≠ null do
5:             child = connector.getFirstChild()
6:             while child ≠ null do
7:                 if not Excluded then
8:                     if not inList then
9:                         add()
10:                        buildDescendant()
11:                    end if
12:                    if connector = XOR then
13:                        remove sibling
14:                    end if
15:                else if mandatory then
16:                    removeParentUntilFirstOptional()
17:                    removeSiblingsChildRequire()
18:                    break
19:                end if
20:                if Parent == XOR then
21:                    remove Sibling child
22:                end if
23:            end while
24:        end while
25:    end if
26: end procedure
```

Algorithm 11 subCheckTransitivity (varVp, fullListExcludes,alreadyChecked)

Data :
Result :

```
1:  if (varVp ≠ null) then
2:      listReq ← null
3:      listExc ← null
4:      listRefElement ← null
5:      if (varVp instanceof Variant) then
6:          listRefElement ← varVp.getListReferencedTables())
7:          listReq ← varVp.getRequire().getReqList()
8:          listExc ← varVp.getExclude().getExclList()
9:      end if
10:     if (varVp instanceof VariationPoint) then
11:         common = varVp.getCommon()
12:         if (common ≠ null) then
13:             listRefElement = common.getListReferencedTables())
14:         end if
15:         listReq ← varVp.getRequire().getReqList()
16:         listExc ← varVp.getExclude().getExclList()
17:     end if
18:     if (listExc ≠ null) then
19:         fullListExcludes.addAll(listExc)
20:         for (s : listExc) do
21:             variant ← Util.getVariantVariationPoint(this, s)
22:             addExcludes(variant, fullListExcludes)
23:             Util.addChildrenVariant_VPoint(variant, fullListExcludes);
24:         end for
25:     end if
26:     stop ← false
27:     if (alreadyChecked ≠ null) then
28:         if (alreadyChecked.contains(varVp.getName())) then
29:             stop ← true
30:         end if
31:     end if
```

```
32:     if (≠ stop) then
33:         alreadyChecked.add(varVp.getName())
34:         if (listReq ≠ null) then
35:             for (req : listReq) do
36:                 if (fullListExcludes ≠ null) then
37:                     if (fullListExcludes.contains(req) ∧ alreadyChecked.size() > 1)
    then
38:                         error ← "TRANSITIVITY"
39:                         listErrors.add(error)
40:                     end if
41:                 end if
42:                 subCheckTransitivity                          ▷ Recursive call
43:             end for
44:         end if
45:         if (listRefElement ≠ null) then
46:             for (ref : listRefElement) do
47:                 if (fullListExcludes ≠ null ∧ alreadyChecked ≠ null) then
48:                     if (fullListExcludes.contains(ref.getName())
    ∧alreadyChecked.size() > 1) then
49:                         error ← "FK"
50:                         listErrors.add(error);
51:                     end if
52:                 end if
53:                 subCheckTransitivity                          ▷ Recursive call
54:             end for
55:         end if
56:     end if
57: end if
```

Algorithm 12 subCheckCircuit

Data :
Result :

```
1: if (varVp ≠ null) then
2:     listReq ← null
3:     listExc ← null
4:     listRefElement ← null
5:     if (varVp instanceof Variant) then
6:         listRefElement ← varVp.getListReferencedTables()
7:         listReq ← varVp.getRequire().getReqList()
8:         listExc ← (varVp).getExclude().getExclList()
9:     end if
10:    if (varVp VariationPoint) then
11:        common ← varVp.getCommon();
12:        if common ≠ null then
13:            listRefElement ← common.getListReferencedTables)
14:        end if
15:        listReq ← varVp.getRequire().getReqList()
16:        listExc ← varVp.getExclude().getExclList()
17:    end if
18:    listExcWithMandatory ← newArrayList < String > ()
19:    listExcWithMandatory.addAll(listExc)
20:    for (s : listExc) do
21:        variant ← Util.getVariantVariationPoint(this, s)
22:        addExcludes(variant, listExcWithMandatory);
23:        Util.addChildrenVariant_VPoint(variant, listExcWithMandatory)
24:    end for
25:    stop ← false
26:    if (alreadyChecked ≠ null) then
27:        if (alreadyChecked.contains(varVp.getName())) then
28:            stop ← true
29:        end if
30:    end if
```

```
31:     if (¬stop) then
32:         alreadyChecked.add(varVp.getName())
33:         if (listExcWithMandatory ≠ null) then
34:             if (listRefElement ≠ null) then
35:                 for (c : listRefElement) do
36:                     listRequiresFK.add(c.getName())
37:                 end for
38:             end if
39:             for (s : listExcWithMandatory) do
40:                 if (alreadyChecked.contains(s)) then
41:                     refV_Vp = ""
42:                     for (v_vp : listRequiresFK) do
43:                         if (alreadyChecked.contains(v_vp)) then
44:                             refV_Vp = refV_Vp + v_vp + ";"
45:                         end if
46:                     end for
47:                     if (¬refV_Vp.equals("")) then
48:                         error ← "CIRCUIT_FK"
49:                         listErrors.add(error)
50:                     else
51:                         error = CIRCUIT
52:                         listErrors.add(error)
53:                     end if
54:                 end if
55:             end for
56:         end if
57:         if (listReq ≠ null) then
58:             for (req : listReq) do
59:                 subCheckCircuit(U.getVVP(this, req), alreadyChecked, listRequiresFK)
60:             end for
61:         end if
62:         if (listRefElement ≠ null) then
63:             for (req : listRefElement) do
64:                 subCheckCircuit(U.getVVP(this, req), alreadyChecked, listRequiresFK)
65:             end for
66:         end if
67:     end if
68: end if
```